Query Processing Architecture Guide

- 02/21/2020
- 89 minutes to read

Applies to: SQL Server (all supported versions) Azure SQL Database

The SQL Server Database Engine processes queries on various data storage architectures such as local tables, partitioned tables, and tables distributed across multiple servers. The following topics cover how SQL Server processes queries and optimizes query reuse through execution plan caching.

Execution modes

The SQL Server Database Engine can process Transact-SQL statements using two distinct processing modes:

- Row mode execution
- Batch mode execution

Row mode execution

Row mode execution is a query processing method used with traditional RDMBS tables, where data is stored in row format. When a query is executed and accesses data in row store tables, the execution tree operators and child operators read each required row, across all the columns specified in the table schema. From each row that is read, SQL Server then retrieves the columns that are required for the result set, as referenced by a SELECT statement, JOIN predicate, or filter predicate.

Note

Row mode execution is very efficient for OLTP scenarios, but can be less efficient when scanning large amounts of data, for example in Data Warehousing scenarios.

Batch mode execution

Batch mode execution is a query processing method used to process multiple rows together (hence the term batch). Each column within a batch is stored as a vector in a separate area of memory, so batch mode processing is vector-based. Batch mode processing also uses algorithms that are optimized for the multi-core CPUs and increased memory throughput that are found on modern hardware.

Batch mode execution is closely integrated with, and optimized around, the columnstore storage format. Batch mode processing operates on compressed data when possible, and eliminates the exchange operator used by row mode execution. The result is better parallelism and faster performance.

When a query is executed in batch mode, and accesses data in columnstore indexes, the execution tree operators and child operators read multiple rows together in column segments. SQL Server reads only the columns required for the result, as referenced by a SELECT statement, JOIN predicate, or filter predicate.
For more information on columnstore indexes, see Columnstore Index Architecture.

Note

Batch mode execution is very efficient Data Warehousing scenarios, where large amounts of data are read and aggregated.

SQL Statement Processing

Processing a single Transact-SQL statement is the most basic way that SQL Server executes Transact-SQL statements. The steps used to process a single SELECT statement that references only local base tables (no views or remote tables) illustrates the basic process.

Logical Operator Precedence

When more than one logical operator is used in a statement, NOT is evaluated first, then AND, and finally OR. Arithmetic, and bitwise, operators are handled before logical operators. For more information, see Operator Precedence.

In the following example, the color condition pertains to product model 21, and not to product model 20, because AND has precedence over OR.

```
SQLCopy
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR ProductModelID = 21
  AND Color = 'Red';
GO
```

You can change the meaning of the query by adding parentheses to force evaluation of the OR first. The following query finds only products under models 20 and 21 that are red.

```
SQLCopy
```

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE (ProductModelID = 20 OR ProductModelID = 21)
  AND Color = 'Red';
GO
```

Using parentheses, even when they are not required, can improve the readability of queries, and reduce the chance of making a subtle mistake because of operator precedence. There is no significant performance penalty in using parentheses. The following example is more readable than the original example, although they are syntactically the same.

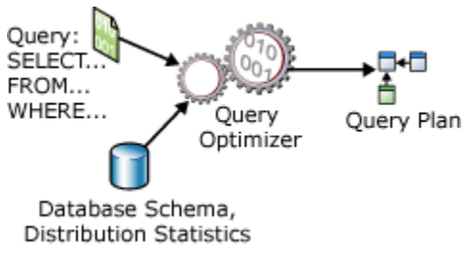
SQLCopy

```
SELECT ProductID, ProductModelID
FROM Production.Product
WHERE ProductModelID = 20 OR (ProductModelID = 21
  AND Color = 'Red');
GO
```

Optimizing SELECT statements

A SELECT statement is non-procedural; it does not state the exact steps that the database server should use to retrieve the requested data. This means that the database server must analyze the statement to determine the most efficient way to extract the requested data. This is referred to as optimizing the SELECT statement. The component that does this is called the Query Optimizer. The input to the Query Optimizer consists of the query, the database schema (table and index definitions), and the database statistics. The output of the Query Optimizer is a query execution plan, sometimes referred to as a query plan, or execution plan. The contents of an execution plan are described in more detail later in this topic.

The inputs and outputs of the Query Optimizer during optimization of a single SELECT statement are illustrated in the following diagram:



A SELECT statement defines only the following:

- The format of the result set. This is specified mostly in the select list. However, other clauses such as ORDER BY and GROUP BY also affect the final form of the result set.
- The tables that contain the source data. This is specified in the FROM clause.
- How the tables are logically related for the purposes of the SELECT statement. This is defined in the join specifications, which may appear in the WHERE clause or in an ON clause following FROM.
- The conditions that the rows in the source tables must satisfy to qualify for the SELECT statement. These are specified in the WHERE and HAVING clauses.

A query execution plan is a definition of the following:

- **The sequence in which the source tables are accessed.** Typically, there are many sequences in which the database server can access the base tables to build the result set. For example, if the SELECT statement references three tables, the database server could first access TableA, use the data from TableA to extract matching rows from TableB, and then use the data from TableB to extract data from TableC. The other sequences in which the database server could access the tables are:
  TableC, TableB, TableA, or
  TableB, TableA, TableC, or
  TableB, TableC, TableA, or
  TableC, TableA, TableB
- **The methods used to extract data from each table.**
  Generally, there are different methods for accessing the data in each table. If only a few rows with specific key values are required, the database server can use an index. If all the rows in the table are required, the database server can ignore the indexes and perform a table scan. If all the rows in a table are required but there is an index whose key columns are in an ORDER BY, performing an index scan instead of a table scan may save a separate sort of the result set. If a table is very small, table scans may be the most efficient method for almost all access to the table.
- **The methods used to compute calculations, and how to filter, aggregate, and sort data from each table.**
  As data is accessed from tables, there are different methods to perform calculations over data such as computing scalar values, and to aggregate and sort data as defined in the query text, for example when using a GROUP BY or ORDER BY clause, and how to filter data, for example when using a WHERE or HAVING clause.

The process of selecting one execution plan from potentially many possible plans is referred to as optimization. The Query Optimizer is one of the most important components of the Database Engine. While some overhead is used by the Query Optimizer to analyze the query and select a plan, this overhead is typically saved several-fold when the Query Optimizer picks an efficient execution plan. For example, two construction companies can be given identical blueprints for a house. If one company spends a few days at the beginning to plan how they will build the house, and the other company begins building without planning, the company that takes the time to plan their project will probably finish first.

The SQL Server Query Optimizer is a cost-based optimizer. Each possible execution plan has an associated cost in terms of the amount of computing resources used. The Query Optimizer must analyze the possible plans and choose the one with the lowest estimated cost. Some complex SELECT statements have thousands of possible execution plans. In these cases, the Query Optimizer does not analyze all possible combinations. Instead, it uses complex algorithms to find an execution plan that has a cost reasonably close to the minimum possible cost.

The SQL Server Query Optimizer does not choose only the execution plan with the lowest resource cost; it chooses the plan that returns results to the user with a reasonable cost in resources and that returns the results the fastest. For example, processing a query in parallel typically uses more resources than processing it serially, but completes the query faster. The SQL Server Query Optimizer will use a parallel execution plan to return results if the load on the server will not be adversely affected.

The SQL Server Query Optimizer relies on distribution statistics when it estimates the resource costs of different methods for extracting information from a table or index. Distribution statistics are kept for columns and indexes, and hold information on the density[1] of the underlying data. This is used to indicate the selectivity of the values in a particular index or column. For example, in a table representing cars, many cars have the same manufacturer, but each car has a unique vehicle identification number (VIN). An index on the VIN is more selective than an index on the manufacturer, because VIN has lower density than manufacturer. If the index statistics are not current, the Query Optimizer may not make the best choice for the current state of the table. For more information about densities, see Statistics.

[1] Density defines the distribution of unique values that exist in the data, or the average number of duplicate values for a given column. As density decreases, selectivity of a value increases.

The SQL Server Query Optimizer is important because it enables the database server to adjust dynamically to changing conditions in the database without requiring input from a programmer or database administrator. This enables programmers to focus on describing the final result of the query. They can trust that the SQL Server Query Optimizer will build an efficient execution plan for the state of the database every time the statement is run.

Note

SQL Server Management Studio has three options to display execution plans:

- The ***Estimated Execution Plan***, which is the compiled plan, as produced by the Query Optimizer.
- The ***Actual Execution Plan***, which is the same as the compiled plan plus its execution context. This includes runtime information available after the execution completes, such as execution warnings, or in newer versions of the Database Engine, the elapsed and CPU time used during execution.
- The ***Live Query Statistics***, which is the same as the compiled plan plus its execution context. This includes runtime information during execution progress, and is updated every second. Runtime information includes for example the actual number of rows flowing through the operators.

Processing a SELECT Statement

The basic steps that SQL Server uses to process a single SELECT statement include the following:

1. The parser scans the SELECT statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
2. A query tree, sometimes referred to as a sequence tree, is built describing the logical steps needed to transform the source data into the format required by the result set.
3. The Query Optimizer analyzes different ways the source tables can be accessed. It then selects the series of steps that return the results fastest while using fewer resources. The query tree is updated to record this exact series of steps. The final, optimized version of the query tree is called the execution plan.
4. The relational engine starts executing the execution plan. As the steps that require data from the base tables are processed, the relational engine requests that the storage engine pass up data from the rowsets requested from the relational engine.
5. The relational engine processes the data returned from the storage engine into the format defined for the result set and returns the result set to the client.

Constant Folding and Expression Evaluation

SQL Server evaluates some constant expressions early to improve query performance. This is referred to as constant folding. A constant is a Transact-SQL literal, such as 3, 'ABC', '2005-12-31', 1.0e3, or 0x12345678.

Foldable Expressions

SQL Server uses constant folding with the following types of expressions:

- Arithmetic expressions, such as 1+1, 5/3*2, that contain only constants.
- Logical expressions, such as 1=1 and 1>2 AND 3>4, that contain only constants.
- Built-in functions that are considered foldable by SQL Server, including CAST and CONVERT. Generally, an intrinsic function is foldable if it is a function of its inputs only and not other contextual information, such as SET options, language settings, database options, and encryption keys. Nondeterministic functions are not foldable. Deterministic built-in functions are foldable, with some exceptions.
- Deterministic methods of CLR user-defined types and deterministic scalar-valued CLR user-defined functions (starting with SQL Server 2012 (11.x)). For more information, see Constant Folding for CLR User-Defined Functions and Methods.

Note

An exception is made for large object types. If the output type of the folding process is a large object type (text,ntext, image, nvarchar(max), varchar(max), varbinary(max), or XML), then SQL Server does not fold the expression.

Nonfoldable Expressions

All other expression types are not foldable. In particular, the following types of expressions are not foldable:

- Nonconstant expressions such as an expression whose result depends on the value of a column.
- Expressions whose results depend on a local variable or parameter, such as @x.
- Nondeterministic functions.
- User-defined Transact-SQL functions[1].
- Expressions whose results depend on language settings.
- Expressions whose results depend on SET options.
- Expressions whose results depend on server configuration options.

[1] Before SQL Server 2012 (11.x), deterministic scalar-valued CLR user-defined functions and methods of CLR user-defined types were not foldable.

Examples of Foldable and Nonfoldable Constant Expressions

Consider the following query:

```
SQLCopy
SELECT *
FROM Sales.SalesOrderHeader AS s
INNER JOIN Sales.SalesOrderDetail AS d
ON s.SalesOrderID = d.SalesOrderID
WHERE TotalDue > 117.00 + 1000.00;
```

If the PARAMETERIZATION database option is not set to FORCED for this query, then the expression 117.00 + 1000.00 is evaluated and replaced by its result, 1117.00, before the query is compiled. Benefits of this constant folding include the following:

- The expression does not have to be evaluated repeatedly at run time.
- The value of the expression after it is evaluated is used by the Query Optimizer to estimate the size of the result set of the portion of the query TotalDue > 117.00 + 1000.00.

On the other hand, if dbo.f is a scalar user-defined function, the expression dbo.f(100) is not folded, because SQL Server does not fold expressions that involve user-defined functions, even if they are deterministic. For more information on parameterization, see Forced Parameterization later in this article.

Expression Evaluation

In addition, some expressions that are not constant folded but whose arguments are known at compile time, whether the arguments are parameters or constants, are evaluated by the result-set size (cardinality) estimator that is part of the optimizer during optimization.

Specifically, the following built-in functions and special operators are evaluated at compile time if all their inputs are known: UPPER, LOWER, RTRIM, DATEPART(YY only), GETDATE, CAST, and CONVERT. The following operators are also evaluated at compile time if all their inputs are known:

- Arithmetic operators: +, -, *, /, unary -
- Logical Operators: AND, OR, NOT
- Comparison operators: <, >, <=, >=, <>, LIKE, IS NULL, IS NOT NULL

No other functions or operators are evaluated by the Query Optimizer during cardinality estimation.

Examples of Compile-Time Expression Evaluation

Consider this stored procedure:

SQLCopy
```
USE AdventureWorks2014;
GO
CREATE PROCEDURE MyProc( @d datetime )
AS
SELECT COUNT(*)
FROM Sales.SalesOrderHeader
WHERE OrderDate > @d+1;
```

During optimization of the SELECT statement in the procedure, the Query Optimizer tries to evaluate the expected cardinality of the result set for the condition OrderDate > @d+1. The expression @d+1 is not constant-folded, because @d is a parameter. However, at optimization time, the value of the parameter is known. This allows the Query Optimizer to accurately estimate the size of the result set, which helps it select a good query plan.

Now consider an example similar to the previous one, except that a local variable @d2 replaces @d+1 in the query and the expression is evaluated in a SET statement instead of in the query.

SQLCopy

```
USE AdventureWorks2014;
GO
CREATE PROCEDURE MyProc2( @d datetime )
AS
BEGIN
  DECLARE @d2 datetime
  SET @d2 = @d+1
  SELECT COUNT(*)
  FROM Sales.SalesOrderHeader
  WHERE OrderDate > @d2
END;
```

When the SELECT statement in *MyProc2* is optimized in SQL Server, the value of @d2 is not known. Therefore, the Query Optimizer uses a default estimate for the selectivity of OrderDate > @d2, (in this case 30 percent).

Processing Other Statements

The basic steps described for processing a SELECT statement apply to other Transact-SQL statements such as INSERT, UPDATE, and DELETE. UPDATE and DELETE statements both have to target the set of rows to be modified or deleted. The process of identifying these rows is the same process used to identify the source rows that contribute to the result set of a SELECT statement. The UPDATE and INSERT statements may both contain embedded SELECT statements that provide the data values to be updated or inserted.

Even Data Definition Language (DDL) statements, such as CREATE PROCEDURE or ALTER TABLE, are ultimately resolved to a series of relational operations on the system catalog tables and sometimes (such as ALTER TABLE ADD COLUMN) against the data tables.

Worktables

The Relational Engine may need to build a worktable to perform a logical operation specified in an Transact-SQL statement. Worktables are internal tables that are used to hold intermediate results. Worktables are generated for certain GROUP BY, ORDER BY, or UNION queries. For example, if an ORDER BY clause references columns that are not covered by any indexes, the Relational Engine may need to generate a worktable to sort the result set into the order requested. Worktables are also sometimes used as spools that temporarily hold the result of executing a part of a query plan. Worktables are built in tempdb and are dropped automatically when they are no longer needed.

View Resolution

The SQL Server query processor treats indexed and nonindexed views differently:

- The rows of an indexed view are stored in the database in the same format as a table. If the Query Optimizer decides to use an indexed view in a query plan, the indexed view is treated the same way as a base table.
- Only the definition of a nonindexed view is stored, not the rows of the view. The Query Optimizer incorporates the logic from the view definition into the execution plan it builds for the Transact-SQL statement that references the nonindexed view.

The logic used by the SQL Server Query Optimizer to decide when to use an indexed view is similar to the logic used to decide when to use an index on a table. If the data in the indexed view covers all or part of the Transact-SQL statement, and the Query Optimizer determines that an index on the view is the low-cost access path, the Query Optimizer will choose the index regardless of whether the view is referenced by name in the query.

When an Transact-SQL statement references a nonindexed view, the parser and Query Optimizer analyze the source of both the Transact-SQL statement and the view and then resolve them into a single execution plan. There is not one plan for the Transact-SQL statement and a separate plan for the view.

For example, consider the following view:

SQLCopy

```
USE AdventureWorks2014;
GO
CREATE VIEW EmployeeName AS
SELECT h.BusinessEntityID, p.LastName, p.FirstName
FROM HumanResources.Employee AS h
JOIN Person.Person AS p
  ON h.BusinessEntityID = p.BusinessEntityID;
GO
```

Based on this view, both of these Transact-SQL statements perform the same operations on the base tables and produce the same results:

SQLCopy

```
/* SELECT referencing the EmployeeName view. */
SELECT LastName AS EmployeeLastName, SalesOrderID, OrderDate
FROM AdventureWorks2014.Sales.SalesOrderHeader AS soh
JOIN AdventureWorks2014.dbo.EmployeeName AS EmpN
  ON (soh.SalesPersonID = EmpN.BusinessEntityID)
```

```
WHERE OrderDate > '20020531';

/* SELECT referencing the Person and Employee tables directly. */
SELECT LastName AS EmployeeLastName, SalesOrderID, OrderDate
FROM AdventureWorks2014.HumanResources.Employee AS e
JOIN AdventureWorks2014.Sales.SalesOrderHeader AS soh
  ON soh.SalesPersonID = e.BusinessEntityID
JOIN AdventureWorks2014.Person.Person AS p
  ON e.BusinessEntityID =p.BusinessEntityID
WHERE OrderDate > '20020531';
```

The SQL Server Management Studio Showplan feature shows that the relational engine builds the same execution plan for both of these SELECT statements.

Using Hints with Views

Hints that are placed on views in a query may conflict with other hints that are discovered when the view is expanded to access its base tables. When this occurs, the query returns an error. For example, consider the following view that contains a table hint in its definition:

SQLCopy

```
USE AdventureWorks2014;
GO
CREATE VIEW Person.AddrState WITH SCHEMABINDING AS
SELECT a.AddressID, a.AddressLine1,
    s.StateProvinceCode, s.CountryRegionCode
FROM Person.Address a WITH (NOLOCK), Person.StateProvince s
WHERE a.StateProvinceID = s.StateProvinceID;
```

Now suppose you enter this query:

SQLCopy

```
SELECT AddressID, AddressLine1, StateProvinceCode, CountryRegionCode
FROM Person.AddrState WITH (SERIALIZABLE)
WHERE StateProvinceCode = 'WA';
```

The query fails, because the hint SERIALIZABLE that is applied on view Person.AddrState in the query is propagated to both tables Person.Address and Person.StateProvince in the view when it is expanded. However, expanding the view also reveals the NOLOCK hint on Person.Address. Because the SERIALIZABLE and NOLOCK hints conflict, the resulting query is incorrect.

The PAGLOCK, NOLOCK, ROWLOCK, TABLOCK, or TABLOCKX table hints conflict with each other, as do the HOLDLOCK, NOLOCK, READCOMMITTED, REPEATABLEREAD, SERIALIZABLE table hints.

Hints can propagate through levels of nested views. For example, suppose a query applies the HOLDLOCK hint on a view v1. When v1 is expanded, we find that view v2 is part of its definition. v2's definition includes a NOLOCK hint on one of its base tables. But this table also inherits the HOLDLOCK hint from the query on view v1. Because the NOLOCK and HOLDLOCK hints conflict, the query fails.

When the FORCE ORDER hint is used in a query that contains a view, the join order of the tables within the view is determined by the position of the view in the ordered construct. For example, the following query selects from three tables and a view:

SQLCopy

```
SELECT * FROM Table1, Table2, View1, Table3
WHERE Table1.Col1 = Table2.Col1
    AND Table2.Col1 = View1.Col1
    AND View1.Col2 = Table3.Col2;
OPTION (FORCE ORDER);
```

And View1 is defined as shown in the following:

SQLCopy

```
CREATE VIEW View1 AS
SELECT Colx, Coly FROM TableA, TableB
WHERE TableA.ColZ = TableB.Colz;
```

The join order in the query plan is Table1, Table2, TableA, TableB, Table3.

Resolving Indexes on Views

As with any index, SQL Server chooses to use an indexed view in its query plan only if the Query Optimizer determines it is beneficial to do so.

Indexed views can be created in any edition of SQL Server. In some editions of some versions of SQL Server, the Query Optimizer automatically considers the indexed view. In some editions of some versions of SQL Server, to use an indexed view, the NOEXPAND table hint must be used. For clarification, see the documentation for each version.

The SQL Server Query Optimizer uses an indexed view when the following conditions are met:

- These session options are set to ON:
  - ANSI_NULLS
  - ANSI_PADDING
  - ANSI_WARNINGS
  - ARITHABORT
  - CONCAT_NULL_YIELDS_NULL
  - QUOTED_IDENTIFIER
- The NUMERIC_ROUNDABORT session option is set to OFF.
- The Query Optimizer finds a match between the view index columns and elements in the query, such as the following:
  - Search condition predicates in the WHERE clause
  - Join operations
  - Aggregate functions
  - GROUP BY clauses
  - Table references
- The estimated cost for using the index has the lowest cost of any access mechanisms considered by the Query Optimizer.
- Every table referenced in the query (either directly, or by expanding a view to access its underlying tables) that corresponds to a table reference in the indexed view must have the same set of hints applied on it in the query.

Note

The READCOMMITTED and READCOMMITTEDLOCK hints are always considered different hints in this context, regardless of the current transaction isolation level.

Other than the requirements for the SET options and table hints, these are the same rules that the Query Optimizer uses to determine whether a table index covers a query. Nothing else has to be specified in the query for an indexed view to be used.

A query does not have to explicitly reference an indexed view in the FROM clause for the Query Optimizer to use the indexed view. If the query contains references to columns in the base tables that are also present in the indexed view, and the Query Optimizer estimates that using the indexed view provides the lowest cost access mechanism, the Query Optimizer chooses the indexed view, similar to the way it chooses base table indexes when they are not directly referenced in a query. The Query Optimizer may choose the view when it contains columns that are not referenced by the query, as long as the view offers the lowest cost option for covering one or more of the columns specified in the query.

The Query Optimizer treats an indexed view referenced in the FROM clause as a standard view. The Query Optimizer expands the definition of the view into the query at the start of the optimization process. Then, indexed view matching is performed. The indexed view may be used in the final execution plan selected by the Query Optimizer, or instead, the plan may materialize necessary data from the view by accessing the base tables referenced by the view. The Query Optimizer chooses the lowest-cost alternative.

Using Hints with Indexed Views

You can prevent view indexes from being used for a query by using the EXPAND VIEWS query hint, or you can use the NOEXPAND table hint to force the use of an index for an indexed view specified in the FROM clause of a query. However, you should let the Query Optimizer dynamically determine the best access methods to use for each query. Limit your use of EXPAND and NOEXPAND to specific cases where testing has shown that they improve performance significantly.

The EXPAND VIEWS option specifies that the Query Optimizer not use any view indexes for the whole query.

When NOEXPAND is specified for a view, the Query Optimizer considers using any indexes defined on the view. NOEXPAND specified with the optional INDEX() clause forces the Query Optimizer to use the specified indexes. NOEXPAND can be specified only for an indexed view and cannot be specified for a view not indexed.

When neither NOEXPAND nor EXPAND VIEWS is specified in a query that contains a view, the view is expanded to access underlying tables. If the query that makes up the view contains any table hints, these hints are propagated to the underlying tables. (This process is explained in more detail in View Resolution.) As long as the set of hints that exists on the underlying tables of the view are identical to each other, the query is eligible to be matched with an indexed view. Most of the time, these hints will match each other, because they are being inherited directly from the view. However, if the query references tables instead of views, and the hints applied directly on these tables are not identical, then such a query is not eligible for matching with an indexed view. If the INDEX, PAGLOCK, ROWLOCK, TABLOCKX, UPDLOCK, or XLOCK hints apply to the tables referenced in the query after view expansion, the query is not eligible for indexed view matching.

If a table hint in the form of INDEX (index_val[,...n]) references a view in a query and you do not also specify the NOEXPAND hint, the index hint is ignored. To specify use of a particular index, use NOEXPAND.

Generally, when the Query Optimizer matches an indexed view to a query, any hints specified on the tables or views in the query are applied directly to the indexed view. If the Query Optimizer chooses not to use an indexed view, any hints are propagated directly to the tables referenced in the view. For more information, see View Resolution. This propagation does not apply to join hints. They are applied only in their original position in the query. Join hints are not considered by the Query Optimizer when matching queries to indexed views. If a query plan uses an indexed view that matches part of a query that contains a join hint, the join hint is not used in the plan.

Hints are not allowed in the definitions of indexed views. In compatibility mode 80 and higher, SQL Server ignores hints inside indexed view definitions when maintaining them, or when executing queries that use indexed views. Although using hints in indexed view definitions will not produce a syntax error in 80 compatibility mode, they are ignored.

Resolving Distributed Partitioned Views

The SQL Server query processor optimizes the performance of distributed partitioned views. The most important aspect of distributed partitioned view performance is minimizing the amount of data transferred between member servers.

SQL Server builds intelligent, dynamic plans that make efficient use of distributed queries to access data from remote member tables:

- The Query Processor first uses OLE DB to retrieve the check constraint definitions from each member table. This allows the query processor to map the distribution of key values across the member tables.
- The Query Processor compares the key ranges specified in an Transact-SQL statement WHERE clause to the map that shows how the rows are distributed in the member tables. The query processor then builds a query execution plan that uses distributed queries to retrieve only those remote rows that are required to complete the Transact-SQL statement. The execution plan is also built in such a way that any access to remote member tables, for either data or metadata, are delayed until the information is required.

For example, consider a system where a customers table is partitioned across Server1 (CustomerID from 1 through 3299999), Server2 (CustomerID from 3300000 through 6599999), and Server3 (CustomerID from 6600000 through 9999999).

Consider the execution plan built for this query executed on Server1:

SQLCopy
```
SELECT *
FROM CompanyData.dbo.Customers
WHERE CustomerID BETWEEN 3200000 AND 3400000;
```

The execution plan for this query extracts the rows with CustomerID key values from 3200000 through 3299999 from the local member table, and issues a distributed query to retrieve the rows with key values from 3300000 through 3400000 from Server2.

The SQL Server Query Processor can also build dynamic logic into query execution plans for Transact-SQL statements in which the key values are not known when the plan must be built. For example, consider this stored procedure:

SQLCopy
```
CREATE PROCEDURE GetCustomer @CustomerIDParameter INT
```

```
AS
SELECT *
FROM CompanyData.dbo.Customers
WHERE CustomerID = @CustomerIDParameter;
```

SQL Server cannot predict what key value will be supplied by the @CustomerIDParameter parameter every time the procedure is executed. Because the key value cannot be predicted, the query processor also cannot predict which member table will have to be accessed. To handle this case, SQL Server builds an execution plan that has conditional logic, referred to as dynamic filters, to control which member table is accessed, based on the input parameter value. Assuming the GetCustomer stored procedure was executed on Server1, the execution plan logic can be represented as shown in the following:

```
SQLCopy
IF @CustomerIDParameter BETWEEN 1 and 3299999
   Retrieve row from local table CustomerData.dbo.Customer_33
ELSE IF @CustomerIDParameter BETWEEN 3300000 and 6599999
   Retrieve row from linked table Server2.CustomerData.dbo.Customer_66
ELSE IF @CustomerIDParameter BETWEEN 6600000 and 9999999
   Retrieve row from linked table Server3.CustomerData.dbo.Customer_99
```

SQL Server sometimes builds these types of dynamic execution plans even for queries that are not parameterized. The Query Optimizer may parameterize a query so that the execution plan can be reused. If the Query Optimizer parameterizes a query referencing a partitioned view, the Query Optimizer can no longer assume the required rows will come from a specified base table. It will then have to use dynamic filters in the execution plan.

Stored Procedure and Trigger Execution

SQL Server stores only the source for stored procedures and triggers. When a stored procedure or trigger is first executed, the source is compiled into an execution plan. If the stored procedure or trigger is again executed before the execution plan is aged from memory, the relational engine detects the existing plan and reuses it. If the plan has aged out of memory, a new plan is built. This process is similar to the process SQL Server follows for all Transact-SQL statements. The main performance advantage that stored procedures and triggers have in SQL Server compared with batches of dynamic Transact-SQL is that their Transact-SQL statements are always the same. Therefore, the relational engine easily matches them with any existing execution plans. Stored procedure and trigger plans are easily reused.

The execution plan for stored procedures and triggers is executed separately from the execution plan for the batch calling the stored procedure or firing the trigger. This allows for greater reuse of the stored procedure and trigger execution plans.

Execution Plan Caching and Reuse

SQL Server has a pool of memory that is used to store both execution plans and data buffers. The percentage of the pool allocated to either execution plans or data buffers fluctuates dynamically, depending on the state of the system. The part of the memory pool that is used to store execution plans is referred to as the plan cache.

The plan cache has two stores for all compiled plans:

- The **Object Plans** cache store (OBJCP) used for plans related to persisted objects (stored procedures, functions, and triggers).
- The **SQL Plans** cache store (SQLCP) used for plans related to autoparameterized, dynamic, or prepared queries.

The query below provides information about memory usage for these two cache stores:

SQLCopy
```
SELECT * FROM sys.dm_os_memory_clerks
WHERE name LIKE '%plans%';
```

Note

The plan cache has two additional stores that are not used for storing plans:

- The **Bound Trees** cache store (PHDR) used for data structures used during plan compilation for views, constraints, and defaults. These structures are known as Bound Trees or Algebrizer Trees.
- The **Extended Stored Procedures** cache store (XPROC) used for predefined system procedures, like sp_executeSql or xp_cmdshell, that are defined using a DLL, not using Transact-SQL statements. The cached structure contains only the function name and the DLL name in which the procedure is implemented.

SQL Server execution plans have the following main components:

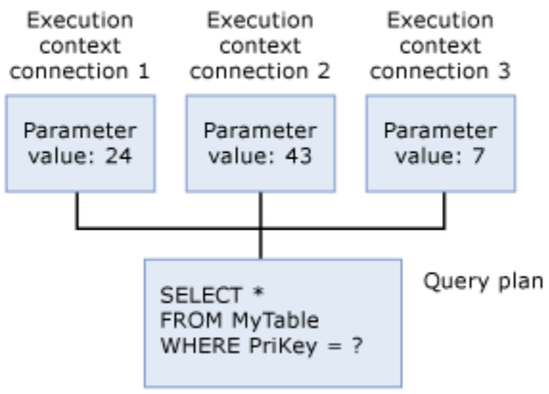
- **Compiled Plan** (or Query Plan)
  The query plan produced by the compilation process is mostly a re-entrant, read-only data structure used by any number of users. It stores information about:
  - Physical operators which implement the operation described by logical operators.
  - The order of these operators, which determines the order in which data is accessed, filtered, and aggregated.
  - The number of estimated rows flowing through the operators.

    Note

    In newer versions of the Database Engine, information about the statistics objects that were used for **Cardinality Estimation** is also stored.

    - What support objects must be created, such as worktables or workfiles in tempdb. No user context or runtime information is stored in the query plan. There are never more than one or two copies of the query plan in memory: one copy for all serial executions and another for all parallel executions. The parallel copy covers all parallel executions, regardless of their degree of parallelism.

- **Execution Context**
  Each user that is currently executing the query has a data structure that holds the data specific to their execution, such as parameter values. This data structure is referred to as the execution context. The execution context data structures are reused, but their content is not. If another user executes the same query, the data structures are reinitialized with the context for the new user.



When any Transact-SQL statement is executed in SQL Server, the Database Engine first looks through the plan cache to verify that an existing execution plan for the same Transact-SQL statement exists. The Transact-SQL statement qualifies as existing if it literally matches a previously executed Transact-SQL statement with a cached plan, character per character. SQL Server reuses any existing plan it finds, saving the overhead of recompiling the Transact-SQL statement. If no execution plan exists, SQL Server generates a new execution plan for the query.

Note

The execution plans for some Transact-SQL statements are not persisted in the plan cache, such as bulk operation statements running on rowstore or statements containing string literals larger than 8 KB in size. These plans only exist while the query is being executed.

SQL Server has an efficient algorithm to find any existing execution plans for any specific Transact-SQL statement. In most systems, the minimal resources that are used by this scan are less than the resources that are saved by being able to reuse existing plans instead of compiling every Transact-SQL statement.

The algorithms to match new Transact-SQL statements to existing, unused execution plans in the plan cache require that all object references be fully qualified. For example, assume that Person is the default schema for the user executing the below SELECT statements. While in this example it is not required that the Person table is fully qualified to execute, it means that the second statement is not matched with an existing plan, but the third is matched:

SQLCopy
```
USE AdventureWorks2014;
GO
SELECT * FROM Person;
GO
SELECT * FROM Person.Person;
GO
SELECT * FROM Person.Person;
GO
```

Changing any of the following SET options for a given execution will affect the ability to reuse plans, because the Database Engine performs constant folding and these options affect the results of such expressions:

ANSI_NULL_DFLT_OFF

FORCEPLAN

ARITHABORT

DATEFIRST

ANSI_PADDING

NUMERIC_ROUNDABORT

ANSI_NULL_DFLT_ON

LANGUAGE

CONCAT_NULL_YIELDS_NULL

DATEFORMAT

ANSI_WARNINGS

QUOTED_IDENTIFIER

ANSI_NULLS

NO_BROWSETABLE

ANSI_DEFAULTS

Caching multiple plans for the same query

Queries and execution plans are uniquely identifiable in the Database Engine, much like a fingerprint:

- The **query plan hash** is a binary hash value calculated on the execution plan for a given query, and used to uniquely identify similar execution plans.
- The **query hash** is a binary hash value calculated on the Transact-SQL text of a query, and is used to uniquely identify queries.

A compiled plan can be retrieved from the plan cache using a **Plan Handle**, which is a transient identifier that remains constant only while the plan remains in the cache. The plan handle is a hash value derived from the compiled plan of the entire batch. The plan handle for a compiled plan remains the same even if one or more statements in the batch get recompiled.

Note

If a plan was compiled for a batch instead of a single statement, the plan for individual statements in the batch can be retrieved using the plan handle and statement offsets.
The sys.dm_exec_requests DMV contains the statement_start_offset and statement_end_offset columns for each record, which refer to the currently executing statement of a currently executing batch or persisted object. For more information, see **sys.dm_exec_requests (Transact-SQL)**.
The sys.dm_exec_query_stats DMV also contains these columns for each record, which refer to the position of a statement within a batch or persisted object. For more information, see **sys.dm_exec_query_stats (Transact-SQL)**.

The actual Transact-SQL text of a batch is stored in a separate memory space from the plan cache, called the **SQL Manager** cache (SQLMGR). The Transact-SQL text for a compiled plan can be retrieved from the sql manager cache using a **SQL Handle**, which is a transient identifier that remains constant only while at least one plan that references it remains in the plan cache. The sql handle is a hash value derived from the entire batch text and is guaranteed to be unique for every batch.

Note

Like a compiled plan, the Transact-SQL text is stored per batch, including the comments. The sql handle contains the MD5 hash of the entire batch text and is guaranteed to be unique for every batch.

The query below provides information about memory usage for the sql manager cache:

SQLCopy
```
SELECT * FROM sys.dm_os_memory_objects
WHERE type = 'MEMOBJ_SQLMGR';
```

There is a 1:N relation between a sql handle and plan handles. Such a condition occurs when the cache key for the compiled plans is different. This may occur due to change in SET options between two executions of the same batch.

Consider the following stored procedure:

SQLCopy
```
USE WideWorldImporters;
GO
CREATE PROCEDURE usp_SalesByCustomer @CID int
AS
SELECT * FROM Sales.Customers
WHERE CustomerID = @CID
GO

SET ANSI_DEFAULTS ON
GO

EXEC usp_SalesByCustomer 10
GO
```

Verify what can be found in the plan cache using the query below:

SQLCopy
```
SELECT cp.memory_object_address, cp.objtype, refcounts, usecounts,
    qs.query_plan_hash, qs.query_hash,
    qs.plan_handle, qs.sql_handle
FROM sys.dm_exec_cached_plans AS cp
```

```
CROSS APPLY sys.dm_exec_sql_text (cp.plan_handle)
CROSS APPLY sys.dm_exec_query_plan (cp.plan_handle)
INNER JOIN sys.dm_exec_query_stats AS qs ON qs.plan_handle = cp.plan_handle
WHERE text LIKE '%usp_SalesByCustomer%'
GO
```

Here is the result set.

Copy

```
memory_object_address    objtype   refcounts   usecounts   query_plan_hash    query_hash
--------------------   -------   ---------   ---------   ------------------ ------------------
0x000001CC6C534060       Proc     2         1         0x3B4303441A1D7E6D 0xA05D5197DA1EAC2D

plan_handle
--------------------------------------------------------------------------------------------
0x0500130095555D02D022F111CD0100000100000000000000000000000000000000000000000000000000000

sql_handle
--------------------------------------------------------------------------------
0x0300130095555D02C864C10061AB0000010000000000000000000000000000000000000000000000000000
```

Now execute the stored procedure with a different parameter, but no other changes to execution context:

SQLCopy

```
EXEC usp_SalesByCustomer 8
GO
```

Verify again what can be found in the plan cache. Here is the result set.

Copy

```
memory_object_address    objtype   refcounts   usecounts   query_plan_hash    query_hash
--------------------   -------   ---------   ---------   ------------------ ------------------
0x000001CC6C534060       Proc     2         2         0x3B4303441A1D7E6D 0xA05D5197DA1EAC2D

plan_handle
--------------------------------------------------------------------------------------------
0x0500130095555D02D022F111CD0100000100000000000000000000000000000000000000000000000000000
```

```
sql_handle
------------------------------------------------------------------------------------
0x0300130095555D02C864C10061AB0000010000000000000000000000000000000000000000000000000000
```

Notice the usecounts has increased to 2, which means the same cached plan was re-used as-is, because the execution context data structures were reused. Now change the SET ANSI_DEFAULTS option and execute the stored procedure using the same parameter.

```
SQLCopy
SET ANSI_DEFAULTS OFF
GO

EXEC usp_SalesByCustomer 8
GO
```

Verify again what can be found in the plan cache. Here is the result set.

```
Copy
memory_object_address    objtype   refcounts   usecounts   query_plan_hash    query_hash
-------------------    -------   ---------   ---------   ------------------ ------------------
0x000001CD01DEC060       Proc      2          1          0x3B4303441A1D7E6D 0xA05D5197DA1EAC2D
0x000001CC6C534060       Proc      2          2          0x3B4303441A1D7E6D 0xA05D5197DA1EAC2D

plan_handle
------------------------------------------------------------------------------------
0x0500130095555D02B031F111CD0100000100000000000000000000000000000000000000000000000000000
0x0500130095555D02D022F111CD0100000100000000000000000000000000000000000000000000000000000

sql_handle
------------------------------------------------------------------------------------
0x0300130095555D02C864C10061AB0000010000000000000000000000000000000000000000000000000000
0x0300130095555D02C864C10061AB0000010000000000000000000000000000000000000000000000000000
```

Notice there are now two entries in the sys.dm_exec_cached_plans DMV output:

- The usecounts column shows the value 1 in the first record which is the plan executed once with SET ANSI_DEFAULTS OFF.
- The usecounts column shows the value 2 in the second record which is the plan executed with SET ANSI_DEFAULTS ON, because it was executed twice.
- The different memory_object_address refers to a different execution plan entry in the plan cache. However, the sql_handle value is the same for both entries because they refer to the same batch.
    - The execution with ANSI_DEFAULTS set to OFF has a new plan_handle, and it's available for reuse for calls that have the same set of SET options. The new plan handle is necessary because the execution context was reinitialized due to changed SET options. But that doesn't trigger a recompile: both entries refer to the same plan and query, as evidenced by the same query_plan_hash and query_hash values.

What this effectively means is that we have two plan entries in the cache corresponding to the same batch, and it underscores the importance of making sure that the plan cache affecting SET options are the same, when the same queries are executed repeatedly, to optimize for plan reuse and keep plan cache size to its required minimum.

Tip

A common pitfall is that different clients may have different default values for the SET options. For example, a connection made through SQL Server Management Studio automatically sets QUOTED_IDENTIFIER to ON, while SQLCMD sets QUOTED_IDENTIFIER to OFF. Executing the same queries from these two clients will result in multiple plans (as described in the example above).

Removing execution plans from the Plan Cache

Execution plans remain in the plan cache as long as there is enough memory to store them. When memory pressure exists, the SQL Server Database Engine uses a cost-based approach to determine which execution plans to remove from the plan cache. To make a cost-based decision, the SQL Server Database Engine increases and decreases a current cost variable for each execution plan according to the following factors.

When a user process inserts an execution plan into the cache, the user process sets the current cost equal to the original query compile cost; for ad-hoc execution plans, the user process sets the current cost to zero. Thereafter, each time a user process references an execution plan, it resets the current cost to the original compile cost; for ad-hoc execution plans the user process increases the current cost. For all plans, the maximum value for the current cost is the original compile cost.

When memory pressure exists, the SQL Server Database Engine responds by removing execution plans from the plan cache. To determine which plans to remove, the SQL Server Database Engine repeatedly examines the state of each execution plan and removes plans when their current cost is zero. An execution plan with zero current cost is not removed automatically when memory pressure exists; it is removed only when the SQL Server Database Engine examines the plan and the current cost is

zero. When examining an execution plan, the SQL Server Database Engine pushes the current cost towards zero by decreasing the current cost if a query is not currently using the plan.

The SQL Server Database Engine repeatedly examines the execution plans until enough have been removed to satisfy memory requirements. While memory pressure exists, an execution plan may have its cost increased and decreased more than once. When memory pressure no longer exists, the SQL Server Database Engine stops decreasing the current cost of unused execution plans and all execution plans remain in the plan cache, even if their cost is zero.

The SQL Server Database Engine uses the resource monitor and user worker threads to free memory from the plan cache in response to memory pressure. The resource monitor and user worker threads can examine plans run concurrently to decrease the current cost for each unused execution plan. The resource monitor removes execution plans from the plan cache when global memory pressure exists. It frees memory to enforce policies for system memory, process memory, resource pool memory, and maximum size for all caches.

The maximum size for all caches is a function of the buffer pool size and cannot exceed the maximum server memory. For more information on configuring the maximum server memory, see the max server memory setting in sp_configure.

The user worker threads remove execution plans from the plan cache when single cache memory pressure exists. They enforce policies for maximum single cache size and maximum single cache entries.

The following examples illustrate which execution plans get removed from the plan cache:

- An execution plan is frequently referenced so that its cost never goes to zero. The plan remains in the plan cache and is not removed unless there is memory pressure and the current cost is zero.
- An ad-hoc execution plan is inserted and is not referenced again before memory pressure exists. Since ad-hoc plans are initialized with a current cost of zero, when the SQL Server Database Engine examines the execution plan, it will see the zero current cost and remove the plan from the plan cache. The ad-hoc execution plan remains in the plan cache with a zero current cost when memory pressure does not exist.

To manually remove a single plan or all plans from the cache, use DBCC FREEPROCCACHE. DBCC FREESYSTEMCACHE can also be used to clear any cache, including plan cache. Starting with SQL Server 2016 (13.x), the ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE to clear the procedure (plan) cache for the database in scope. A change in some configuration settings via sp_configure and reconfigure will also cause plans to be removed from plan cache. You can find the list of these configuration settings in the Remarks section of the DBCC FREEPROCCACHE article. A configuration change like this will log the following informational message in the error log:

SQL Server has encountered %d occurrence(s) of cachestore flush for the '%s' cachestore (part of plan cache) due to some database maintenance or reconfigure operations.

Recompiling Execution Plans

Certain changes in a database can cause an execution plan to be either inefficient or invalid, based on the new state of the database. SQL Server detects the changes that invalidate an execution plan and marks the plan as not valid. A new plan must then be recompiled for the next connection that executes the query. The conditions that invalidate a plan include the following:

- Changes made to a table or view referenced by the query (ALTER TABLE and ALTER VIEW).
- Changes made to a single procedure, which would drop all plans for that procedure from the cache (ALTER PROCEDURE).
- Changes to any indexes used by the execution plan.
- Updates on statistics used by the execution plan, generated either explicitly from a statement, such as UPDATE STATISTICS, or generated automatically.
- Dropping an index used by the execution plan.
- An explicit call to sp_recompile.
- Large numbers of changes to keys (generated by INSERT or DELETE statements from other users that modify a table referenced by the query).
- For tables with triggers, if the number of rows in the inserted or deleted tables grows significantly.
- Executing a stored procedure using the WITH RECOMPILE option.

Most recompilations are required either for statement correctness or to obtain potentially faster query execution plans.

In SQL Server versions prior to 2005, whenever a statement within a batch causes recompilation, the entire batch, whether submitted through a stored procedure, trigger, ad-hoc batch, or prepared statement, was recompiled. Starting with SQL Server 2005 (9.x), only the statement inside the batch that triggers recompilation is recompiled. Also, there are additional types of recompilations in SQL Server 2005 (9.x) and later because of its expanded feature set.

Statement-level recompilation benefits performance because, in most cases, a small number of statements causes recompilations and their associated penalties, in terms of CPU time and locks. These penalties are therefore avoided for the other statements in the batch that do not have to be recompiled.

The sql_statement_recompile extended event (xEvent) reports statement-level recompilations. This xEvent occurs when a statement-level recompilation is required by any kind of batch. This includes stored procedures, triggers, ad hoc batches and queries. Batches may be submitted through several interfaces, including sp_executesql, dynamic SQL, Prepare methods or Execute methods. The recompile_cause column of sql_statement_recompile xEvent contains an integer code that indicates the reason for the recompilation. The following table contains the possible reasons:

Schema changed

Statistics changed

Deferred compile

SET option changed

Temporary table changed

Remote rowset changed

FOR BROWSE permission changed

Query notification environment changed

Partitioned view changed

Cursor options changed

OPTION (RECOMPILE) requested

Parameterized plan flushed

Plan affecting database version changed

Query Store plan forcing policy changed

Query Store plan forcing failed

Query Store missing the plan

Note

In SQL Server versions where xEvents are not available, then the SQL Server Profiler **SP:Recompile** trace event can be used for the same purpose of reporting statement-level recompilations. The trace event SQL:StmtRecompile also reports statement-level recompilations, and this trace event can also be used to track and debug recompilations.
Whereas SP:Recompile generates only for stored procedures and triggers, SQL:StmtRecompile generates for stored procedures, triggers, ad-hoc batches, batches that are executed by using sp_executesql, prepared queries, and dynamic SQL.
The *EventSubClass* column of SP:Recompile and SQL:StmtRecompile contains an integer code that indicates the reason for the recompilation. The codes are described **here**.

Note

When the AUTO_UPDATE_STATISTICS database option is set to ON, queries are recompiled when they target tables or indexed views whose statistics have been updated or whose cardinalities have changed significantly since the last execution. This behavior applies to standard user-defined tables, temporary tables, and the inserted and deleted tables created by DML triggers. If query performance is affected by excessive recompilations, consider changing this setting to OFF. When the AUTO_UPDATE_STATISTICS database option is set to OFF, no recompilations occur based on statistics or cardinality changes, with the exception of the inserted and deleted tables that are created by DML INSTEAD OF triggers. Because these tables are created in tempdb, the recompilation of queries that access them depends on the setting of AUTO_UPDATE_STATISTICS in tempdb. Note that in SQL Server prior to 2005, queries continue to recompile based on cardinality changes to the DML trigger inserted and deleted tables, even when this setting is OFF.

Parameters and Execution Plan Reuse

The use of parameters, including parameter markers in ADO, OLE DB, and ODBC applications, can increase the reuse of execution plans.

Warning

Using parameters or parameter markers to hold values that are typed by end users is more secure than concatenating the values into a string that is then executed by using either a data access API method, the EXECUTE statement, or the sp_executesql stored procedure.

The only difference between the following two SELECT statements is the values that are compared in the WHERE clause:

SQLCopy
```
SELECT *
FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
```
SQLCopy
```
SELECT *
FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 4;
```

The only difference between the execution plans for these queries is the value stored for the comparison against the ProductSubcategoryID column. While the goal is for SQL Server to always recognize that the statements generate essentially the same plan and reuse the plans, SQL Server sometimes does not detect this in complex Transact-SQL statements.

Separating constants from the Transact-SQL statement by using parameters helps the relational engine recognize duplicate plans. You can use parameters in the following ways:

- In Transact-SQL , use sp_executesql:

  SQLCopy
  ```
  DECLARE @MyIntParm INT
  SET @MyIntParm = 1
  EXEC sp_executesql
    N'SELECT *
    FROM AdventureWorks2014.Production.Product
    WHERE ProductSubcategoryID = @Parm',
    N'@Parm INT',
    @MyIntParm
  ```

  This method is recommended for Transact-SQL scripts, stored procedures, or triggers that generate SQL statements dynamically.

- ADO, OLE DB, and ODBC use parameter markers. Parameter markers are question marks (?) that replace a constant in an SQL statement and are bound to a program variable. For example, you would do the following in an ODBC application:
  - Use SQLBindParameter to bind an integer variable to the first parameter marker in an SQL statement.
  - Put the integer value in the variable.
  - Execute the statement, specifying the parameter marker (?):

  Copy
  ```
  SQLExecDirect(hstmt,
    "SELECT *
    FROM AdventureWorks2014.Production.Product
    WHERE ProductSubcategoryID = ?",
    SQL_NTS);
  ```

  The SQL Server Native Client OLE DB Provider and the SQL Server Native Client ODBC driver included with SQL Server use sp_executesql to send statements to SQL Server when parameter markers are used in applications.

- To design stored procedures, which use parameters by design.

If you do not explicitly build parameters into the design of your applications, you can also rely on the SQL Server Query Optimizer to automatically parameterize certain queries by using the default behavior of simple parameterization. Alternatively,

you can force the Query Optimizer to consider parameterizing all queries in the database by setting the PARAMETERIZATION option of the ALTER DATABASE statement to FORCED.

When forced parameterization is enabled, simple parameterization can still occur. For example, the following query cannot be parameterized according to the rules of forced parameterization:

SQLCopy
```
SELECT * FROM Person.Address
WHERE AddressID = 1 + 2;
```

However, it can be parameterized according to simple parameterization rules. When forced parameterization is tried but fails, simple parameterization is still subsequently tried.

Simple Parameterization

In SQL Server, using parameters or parameter markers in Transact-SQL statements increases the ability of the relational engine to match new Transact-SQL statements with existing, previously-compiled execution plans.

Warning

Using parameters or parameter markers to hold values typed by end users is more secure than concatenating the values into a string that is then executed using either a data access API method, the EXECUTE statement, or the sp_executesql stored procedure.

If a Transact-SQL statement is executed without parameters, SQL Server parameterizes the statement internally to increase the possibility of matching it against an existing execution plan. This process is called simple parameterization. In SQL Server versions prior to 2005, the process was referred to as auto-parameterization.

Consider this statement:

SQLCopy
```
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
```

The value 1 at the end of the statement can be specified as a parameter. The relational engine builds the execution plan for this batch as if a parameter had been specified in place of the value 1. Because of this simple parameterization, SQL Server recognizes that the following two statements generate essentially the same execution plan and reuses the first plan for the second statement:

SQLCopy
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 1;
SQLCopy
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductSubcategoryID = 4;

When processing complex Transact-SQL statements, the relational engine may have difficulty determining which expressions can be parameterized. To increase the ability of the relational engine to match complex Transact-SQL statements to existing, unused execution plans, explicitly specify the parameters using either sp_executesql or parameter markers.

Note

When the +, -, *, /, or % arithmetic operators are used to perform implicit or explicit conversion of int, smallint, tinyint, or bigint constant values to the float, real, decimal or numeric data types, SQL Server applies specific rules to calculate the type and precision of the expression results. However, these rules differ, depending on whether the query is parameterized or not. Therefore, similar expressions in queries can, in some cases, produce differing results.

Under the default behavior of simple parameterization, SQL Server parameterizes a relatively small class of queries. However, you can specify that all queries in a database be parameterized, subject to certain limitations, by setting the PARAMETERIZATION option of the ALTER DATABASE command to FORCED. Doing so may improve the performance of databases that experience high volumes of concurrent queries by reducing the frequency of query compilations.

Alternatively, you can specify that a single query, and any others that are syntactically equivalent but differ only in their parameter values, be parameterized.

Forced Parameterization

You can override the default simple parameterization behavior of SQL Server by specifying that all SELECT, INSERT, UPDATE, and DELETE statements in a database be parameterized, subject to certain limitations. Forced parameterization is enabled by setting the PARAMETERIZATION option to FORCED in the ALTER DATABASE statement. Forced parameterization may improve the performance of certain databases by reducing the frequency of query compilations and recompilations. Databases that may benefit from forced parameterization are generally those that experience high volumes of concurrent queries from sources such as point-of-sale applications.

When the PARAMETERIZATION option is set to FORCED, any literal value that appears in a SELECT, INSERT, UPDATE, or DELETE statement, submitted in any form, is converted to a parameter during query compilation. The exceptions are literals that appear in the following query constructs:

- INSERT...EXECUTE statements.
- Statements inside the bodies of stored procedures, triggers, or user-defined functions. SQL Server already reuses query plans for these routines.
- Prepared statements that have already been parameterized on the client-side application.
- Statements that contain XQuery method calls, where the method appears in a context where its arguments would typically be parameterized, such as a WHERE clause. If the method appears in a context where its arguments would not be parameterized, the rest of the statement is parameterized.
- Statements inside a Transact-SQL cursor. (SELECT statements inside API cursors are parameterized.)
- Deprecated query constructs.
- Any statement that is run in the context of ANSI_PADDING or ANSI_NULLS set to OFF.
- Statements that contain more than 2,097 literals that are eligible for parameterization.
- Statements that reference variables, such as WHERE T.col2 >= @bb.
- Statements that contain the RECOMPILE query hint.
- Statements that contain a COMPUTE clause.
- Statements that contain a WHERE CURRENT OF clause.

Additionally, the following query clauses are not parameterized. Note that in these cases, only the clauses are not parameterized. Other clauses within the same query may be eligible for forced parameterization.

- The <select_list> of any SELECT statement. This includes SELECT lists of subqueries and SELECT lists inside INSERT statements.
- Subquery SELECT statements that appear inside an IF statement.
- The TOP, TABLESAMPLE, HAVING, GROUP BY, ORDER BY, OUTPUT...INTO, or FOR XML clauses of a query.
- Arguments, either direct or as subexpressions, to OPENROWSET, OPENQUERY, OPENDATASOURCE, OPENXML, or any FULLTEXT operator.
- The pattern and escape_character arguments of a LIKE clause.
- The style argument of a CONVERT clause.
- Integer constants inside an IDENTITY clause.
- Constants specified by using ODBC extension syntax.
- Constant-foldable expressions that are arguments of the +, -, *, /, and % operators. When considering eligibility for forced parameterization, SQL Server considers an expression to be constant-foldable when either of the following conditions is true:
    - No columns, variables, or subqueries appear in the expression.
    - The expression contains a CASE clause.
- Arguments to query hint clauses. These include the *number_of_rows* argument of the FAST query hint, the *number_of_processors* argument of the MAXDOP query hint, and the *number* argument of the MAXRECURSION query hint.

Parameterization occurs at the level of individual Transact-SQL statements. In other words, individual statements in a batch are parameterized. After compiling, a parameterized query is executed in the context of the batch in which it was originally submitted. If an execution plan for a query is cached, you can determine whether the query was parameterized by referencing the sql column of the sys.syscacheobjects dynamic management view. If a query is parameterized, the names and data types of parameters come before the text of the submitted batch in this column, such as (@1 tinyint).

Note

Parameter names are arbitrary. Users or applications should not rely on a particular naming order. Also, the following can change between versions of SQL Server and Service Pack upgrades: Parameter names, the choice of literals that are parameterized, and the spacing in the parameterized text.

Data Types of Parameters

When SQL Server parameterizes literals, the parameters are converted to the following data types:

- Integer literals whose size would otherwise fit within the int data type parameterize to int. Larger integer literals that are parts of predicates that involve any comparison operator (includes <, <=, =, !=, >, >=, , !<, !>, <>, ALL, ANY, SOME, BETWEEN, and IN) parameterize to numeric(38,0). Larger literals that are not parts of predicates that involve comparison operators parameterize to numeric whose precision is just large enough to support its size and whose scale is 0.
- Fixed-point numeric literals that are parts of predicates that involve comparison operators parameterize to numeric whose precision is 38 and whose scale is just large enough to support its size. Fixed-point numeric literals that are not parts of predicates that involve comparison operators parameterize to numeric whose precision and scale are just large enough to support its size.
- Floating point numeric literals parameterize to float(53).
- Non-Unicode string literals parameterize to varchar(8000) if the literal fits within 8,000 characters, and to varchar(max) if it is larger than 8,000 characters.
- Unicode string literals parameterize to nvarchar(4000) if the literal fits within 4,000 Unicode characters, and to nvarchar(max) if the literal is larger than 4,000 characters.
- Binary literals parameterize to varbinary(8000) if the literal fits within 8,000 bytes. If it is larger than 8,000 bytes, it is converted to varbinary(max).
- Money type literals parameterize to money.

Guidelines for Using Forced Parameterization

Consider the following when you set the PARAMETERIZATION option to FORCED:

- Forced parameterization, in effect, changes the literal constants in a query to parameters when compiling a query. Therefore, the Query Optimizer might choose suboptimal plans for queries. In particular, the Query Optimizer is less likely to match the query to an indexed view or an index on a computed column. It may also choose suboptimal plans for queries posed on partitioned tables and distributed partitioned views. Forced parameterization should not be used for environments that rely heavily on indexed views and indexes on computed columns. Generally, the PARAMETERIZATION FORCED option should only be used by experienced database administrators after determining that doing this does not adversely affect performance.
- Distributed queries that reference more than one database are eligible for forced parameterization as long as the PARAMETERIZATION option is set to FORCED in the database whose context the query is running.
- Setting the PARAMETERIZATION option to FORCED flushes all query plans from the plan cache of a database, except those that currently are compiling, recompiling, or running. Plans for queries that are compiling or running during the setting change are parameterized the next time the query is executed.
- Setting the PARAMETERIZATION option is an online operation that it requires no database-level exclusive locks.
- The current setting of the PARAMETERIZATION option is preserved when reattaching or restoring a database.

You can override the behavior of forced parameterization by specifying that simple parameterization be attempted on a single query, and any others that are syntactically equivalent but differ only in their parameter values. Conversely, you can specify that forced parameterization be attempted on only a set of syntactically equivalent queries, even if forced parameterization is disabled in the database. Plan guides are used for this purpose.

Note

When the PARAMETERIZATION option is set to FORCED, the reporting of error messages may differ from when the PARAMETERIZATION option is set to SIMPLE: multiple error messages may be reported under forced parameterization, where fewer messages would be reported under simple parameterization, and the line numbers in which errors occur may be reported incorrectly.

Preparing SQL Statements

The SQL Server relational engine introduces full support for preparing Transact-SQL statements before they are executed. If an application has to execute an Transact-SQL statement several times, it can use the database API to do the following:

- Prepare the statement once. This compiles the Transact-SQL statement into an execution plan.
- Execute the precompiled execution plan every time it has to execute the statement. This prevents having to recompile the Transact-SQL statement on each execution after the first time.
  Preparing and executing statements is controlled by API functions and methods. It is not part of the Transact-SQL language. The prepare/execute model of executing Transact-SQL statements is supported by the SQL Server Native Client OLE DB Provider and the SQL Server Native Client ODBC driver. On a prepare request, either the provider or the driver sends the statement to SQL Server with a request to prepare the statement. SQL Server compiles an

execution plan and returns a handle for that plan to the provider or driver. On an execute request, either the provider or the driver sends the server a request to execute the plan that is associated with the handle.

Prepared statements cannot be used to create temporary objects on SQL Server. Prepared statements cannot reference system stored procedures that create temporary objects, such as temporary tables. These procedures must be executed directly.

Excess use of the prepare/execute model can degrade performance. If a statement is executed only once, a direct execution requires only one network round-trip to the server. Preparing and executing an Transact-SQL statement executed only one time requires an extra network round-trip; one trip to prepare the statement and one trip to execute it.

Preparing a statement is more effective if parameter markers are used. For example, assume that an application is occasionally asked to retrieve product information from the AdventureWorks sample database. There are two ways the application can do this.

Using the first way, the application can execute a separate query for each product requested:

```
SQLCopy
SELECT * FROM AdventureWorks2014.Production.Product
WHERE ProductID = 63;
```

Using the second way, the application does the following:

1. Prepares a statement that contains a parameter marker (?):
   ```
   SQLCopy
   SELECT * FROM AdventureWorks2014.Production.Product
   WHERE ProductID = ?;
   ```
2. Binds a program variable to the parameter marker.
3. Each time product information is needed, fills the bound variable with the key value and executes the statement.

The second way is more efficient when the statement is executed more than three times.

In SQL Server, the prepare/execute model has no significant performance advantage over direct execution, because of the way SQL Server reuses execution plans. SQL Server has efficient algorithms for matching current Transact-SQL statements with execution plans that are generated for prior executions of the same Transact-SQL statement. If an application executes a Transact-SQL statement with parameter markers multiple times, SQL Server will reuse the execution plan from the first execution for the second and subsequent executions (unless the plan ages from the plan cache). The prepare/execute model still has these benefits:

- Finding an execution plan by an identifying handle is more efficient than the algorithms used to match an Transact-SQL statement to existing execution plans.
- The application can control when the execution plan is created and when it is reused.
- The prepare/execute model is portable to other databases, including earlier versions of SQL Server.

Parameter Sensitivity

Parameter sensitivity, also known as "parameter sniffing", refers to a process whereby SQL Server "sniffs" the current parameter values during compilation or recompilation, and passes it along to the Query Optimizer so that they can be used to generate potentially more efficient query execution plans.

Parameter values are sniffed during compilation or recompilation for the following types of batches:

- Stored procedures
- Queries submitted via sp_executesql
- Prepared queries

For more information on troubleshooting bad parameter sniffing issues, see Troubleshoot queries with parameter-sensitive query execution plan issues.

Note

For queries using the RECOMPILE hint, both parameter values and current values of local variables are sniffed. The values sniffed (of parameters and local variables) are those that exist at the place in the batch just before the statement with the RECOMPILE hint. In particular, for parameters, the values that came along with the batch invocation call are not sniffed.

Parallel Query Processing

SQL Server provides parallel queries to optimize query execution and index operations for computers that have more than one microprocessor (CPU). Because SQL Server can perform a query or index operation in parallel by using several operating system worker threads, the operation can be completed quickly and efficiently.

During query optimization, SQL Server looks for queries or index operations that might benefit from parallel execution. For these queries, SQL Server inserts exchange operators into the query execution plan to prepare the query for parallel execution. An exchange operator is an operator in a query execution plan that provides process management, data redistribution, and flow control. The exchange operator includes the Distribute Streams, Repartition Streams, and Gather Streams logical operators as subtypes, one or more of which can appear in the Showplan output of a query plan for a parallel query.

Important

Certain constructs inhibit SQL Server's ability to leverage parallelism on the entire execution plan, or parts or the execution plan.

Constructs that inhibit parallelism include:

- **Scalar UDFs**
  For more information on scalar user-defined functions, see Create User-defined Functions. Starting with SQL Server 2019 (15.x), the SQL Server Database Engine has the ability to inline these functions, and unlock use of parallelism during query processing. For more information on scalar UDF inlining, see Intelligent query processing in SQL databases.
- **Remote Query**
  For more information on Remote Query, see Showplan Logical and Physical Operators Reference.
- **Dynamic cursors**
  For more information on cursors, see DECLARE CURSOR.
- **Recursive queries**
  For more information on recursion, see Guidelines for Defining and Using Recursive Common Table Expressions and Recursion in T-SQL.
- **Multi-statement table-valued functions (MSTVFs)**
  For more information on MSTVFs, see Create User-defined Functions (Database Engine).
- **TOP keyword**
  For more information, see TOP (Transact-SQL).

A query execution plan may contain the **NonParallelPlanReason** attribute in the **QueryPlan** element which describes why parallelism was not used. Values for this attribute include:

TABLE 1

NonParallelPlanReason Value	Description
MaxDOPSetToOne	Maximum degree of parallelism set to 1.
EstimatedDOPIsOne	Estimated degree of parallelism is 1.
NoParallelWithRemoteQuery	Parallelism is not supported for remote queries.
NoParallelDynamicCursor	Parallel plans not supported for dynamic cursors.
NoParallelFastForwardCursor	Parallel plans not supported for fast forward cursors.
NoParallelCursorFetchByBookmark	Parallel plans not supported for cursors that fetch by bookmark.

TABLE 1

NonParallelPlanReason Value	Description
NoParallelCreateIndexInNonEnterpriseEdition	Parallel index creation not supported for non-Enterprise edition.
NoParallelPlansInDesktopOrExpressEdition	Parallel plans not supported for Desktop and Express edition.
NonParallelizableIntrinsicFunction	Query is referencing a non-parallelizable intrinsic function.
CLRUserDefinedFunctionRequiresDataAccess	Parallelism not supported for a CLR UDF that requires data access.
TSQLUserDefinedFunctionsNotParallelizable	Query is referencing a T-SQL User Defined Function that was not parallelizable.
TableVariableTransactionsDoNotSupportParallelNestedTransaction	Table variable transactions do not support parallel nested transactions.
DMLQueryReturnsOutputToClient	DML query returns output to client and is not parallelizable.
MixedSerialAndParallelOnlineIndexBuildNotSupported	Unsupported mix of serial and parallel plans for a single online index build.
CouldNotGenerateValidParallelPlan	Verifying parallel plan failed, failing back to serial.
NoParallelForMemoryOptimizedTables	Parallelism not supported for referenced In-Memory OLTP tables.
NoParallelForDmlOnMemoryOptimizedTable	Parallelism not supported for DML on an In-Memory OLTP table.
NoParallelForNativelyCompiledModule	Parallelism not supported for referenced natively compiled modules.
NoRangesResumableCreate	Range generation failed for a resumable create operation.

After exchange operators are inserted, the result is a parallel-query execution plan. A parallel-query execution plan can use more than one worker thread. A serial execution plan, used by a non-parallel (serial) query, uses only one worker thread for its execution. The actual number of worker threads used by a parallel query is determined at query plan execution initialization and is determined by the complexity of the plan and the degree of parallelism.

Degree of parallelism (DOP) determines the maximum number of CPUs that are being used; it does not mean the number of worker threads that are being used. The DOP limit is set per task. It is not a per request or per query limit. This means that during a parallel query execution, a single request can spawn multiple tasks which are assigned to a scheduler. More processors than specified by the MAXDOP may be used concurrently at any given point of query execution, when different tasks are executed concurrently. For more information, see the Thread and Task Architecture Guide.

The SQL Server Query Optimizer does not use a parallel execution plan for a query if any one of the following conditions is true:

- The serial execution cost of the query is not high enough to consider an alternative, parallel execution plan.
- A serial execution plan is considered faster than any possible parallel execution plan for the particular query.
- The query contains scalar or relational operators that cannot be run in parallel. Certain operators can cause a section of the query plan to run in serial mode, or the whole plan to run in serial mode.

Degree of Parallelism

SQL Server automatically detects the best degree of parallelism for each instance of a parallel query execution or index data definition language (DDL) operation. It does this based on the following criteria:

1. Whether SQL Server is **running on a computer that has more than one microprocessor or CPU**, such as a symmetric multiprocessing computer (SMP). Only computers that have more than one CPU can use parallel queries.
2. Whether **sufficient worker threads are available**. Each query or index operation requires a certain number of worker threads to execute. Executing a parallel plan requires more worker threads than a serial plan, and the number of required worker threads increases with the degree of parallelism. When the worker thread requirement of the parallel plan for a specific degree of parallelism cannot be satisfied, the SQL Server Database Engine decreases the degree of parallelism automatically or completely abandons the parallel plan in the specified workload context. It then executes the serial plan (one worker thread).
3. The **type of query or index operation executed**. Index operations that create or rebuild an index, or drop a clustered index and queries that use CPU cycles heavily are the best candidates for a parallel plan. For example, joins of large tables, large aggregations, and sorting of large result sets are good candidates. Simple queries, frequently found in transaction processing applications, find the additional coordination required to execute a query in parallel outweigh the potential performance boost. To distinguish between queries that benefit from parallelism and those that do not benefit, the SQL Server Database Engine compares the estimated cost of executing the query or index operation with the cost threshold for parallelism value. Users can change the default value of 5 using sp_configure if proper testing found that a different value is better suited for the running workload.
4. Whether there are a **sufficient number of rows to process**. If the Query Optimizer determines that the number of rows is too low, it does not introduce exchange operators to distribute the rows. Consequently, the operators are executed serially. Executing the operators in a serial plan avoids scenarios when the startup, distribution, and coordination costs exceed the gains achieved by parallel operator execution.

5. Whether **current distribution statistics are available**. If the highest degree of parallelism is not possible, lower degrees are considered before the parallel plan is abandoned. For example, when you create a clustered index on a view, distribution statistics cannot be evaluated, because the clustered index does not yet exist. In this case, the SQL Server Database Engine cannot provide the highest degree of parallelism for the index operation. However, some operators, such as sorting and scanning, can still benefit from parallel execution.

Note

Parallel index operations are only available in SQL Server Enterprise, Developer, and Evaluation editions.

At execution time, the SQL Server Database Engine determines whether the current system workload and configuration information previously described allow for parallel execution. If parallel execution is warranted, the SQL Server Database Engine determines the optimal number of worker threads and spreads the execution of the parallel plan across those worker threads. When a query or index operation starts executing on multiple worker threads for parallel execution, the same number of worker threads is used until the operation is completed. The SQL Server Database Engine re-examines the optimal number of worker thread decisions every time an execution plan is retrieved from the plan cache. For example, one execution of a query can result in the use of a serial plan, a later execution of the same query can result in a parallel plan using three worker threads, and a third execution can result in a parallel plan using four worker threads.

The update and delete operators in a parallel query execution plan are executed serially, but the WHERE clause of an UPDATE or a DELETE statement may be executed in parallel. The actual data changes are then serially applied to the database.

Up to SQL Server 2012 (11.x), the insert operator is also executed serially. However, the SELECT part of an INSERT statement may be executed in parallel. The actual data changes are then serially applied to the database.

Starting with SQL Server 2014 (12.x) and database compatibility level 110, the SELECT … INTO statement can be executed in parallel. Other forms of insert operators work the same way as described for SQL Server 2012 (11.x).

Starting with SQL Server 2016 (13.x) and database compatibility level 130, the INSERT … SELECT statement can be executed in parallel when inserting into heaps or clustered columnstore indexes (CCI), and using the TABLOCK hint. Inserts into local temporary tables (identified by the # prefix) and global temporary tables (identified by ## prefixes) are also enabled for parallelism using the TABLOCK hint. For more information, see INSERT (Transact-SQL).

Static and keyset-driven cursors can be populated by parallel execution plans. However, the behavior of dynamic cursors can be provided only by serial execution. The Query Optimizer always generates a serial execution plan for a query that is part of a dynamic cursor.

Overriding Degrees of Parallelism

The degree of parallelism sets the number of processors to use in parallel plan execution. This configuration can be set at various levels:

1. Server level, using the **max degree of parallelism (MAXDOP)** server configuration option.
   Applies to: SQL Server

   Note

   SQL Server 2019 (15.x) introduces automatic recommendations for setting the MAXDOP server configuration option during the installation process. The setup user interface allows you to either accept the recommended settings or enter your own value. For more information, see **Database Engine Configuration - MaxDOP page**.

2. Workload level, using the **MAX_DOP** Resource Governor workload group configuration option.
   Applies to: SQL Server
3. Database level, using the **MAXDOP** database scoped configuration.
   Applies to: SQL Server and Azure SQL Database
4. Query or index statement level, using the **MAXDOP** query hint or **MAXDOP** index option. For example, you can use the MAXDOP option to control, by increasing or reducing, the number of processors dedicated to an online index operation. In this way, you can balance the resources used by an index operation with those of the concurrent users.
   Applies to: SQL Server and Azure SQL Database

Setting the max degree of parallelism option to 0 (default) enables SQL Server to use all available processors up to a maximum of 64 processors in a parallel plan execution. Although SQL Server sets a runtime target of 64 logical processors when MAXDOP option is set to 0, a different value can be manually set if needed. Setting MAXDOP to 0 for queries and indexes allows SQL Server to use all available processors up to a maximum of 64 processors for the given queries or indexes in a parallel plan execution. MAXDOP is not an enforced value for all parallel queries, but rather a tentative target for all queries eligible for parallelism. This means that if not enough worker threads are available at runtime, a query may execute with a lower degree of parallelism than the MAXDOP server configuration option.

Tip

Refer to this **documentation page** for guidelines on configuring MAXDOP.

Parallel Query Example

The following query counts the number of orders placed in a specific quarter, starting on April 1, 2000, and in which at least one line item of the order was received by the customer later than the committed date. This query lists the count of such orders grouped by each order priority and sorted in ascending priority order.

This example uses theoretical table and column names.

SQLCopy
```
SELECT o_orderpriority, COUNT(*) AS Order_Count
FROM orders
WHERE o_orderdate >= '2000/04/01'
  AND o_orderdate < DATEADD (mm, 3, '2000/04/01')
  AND EXISTS
      (
       SELECT *
        FROM    lineitem
        WHERE l_orderkey = o_orderkey
          AND l_commitdate < l_receiptdate
      )
  GROUP BY o_orderpriority
  ORDER BY o_orderpriority
```

Assume the following indexes are defined on the lineitem and orders tables:

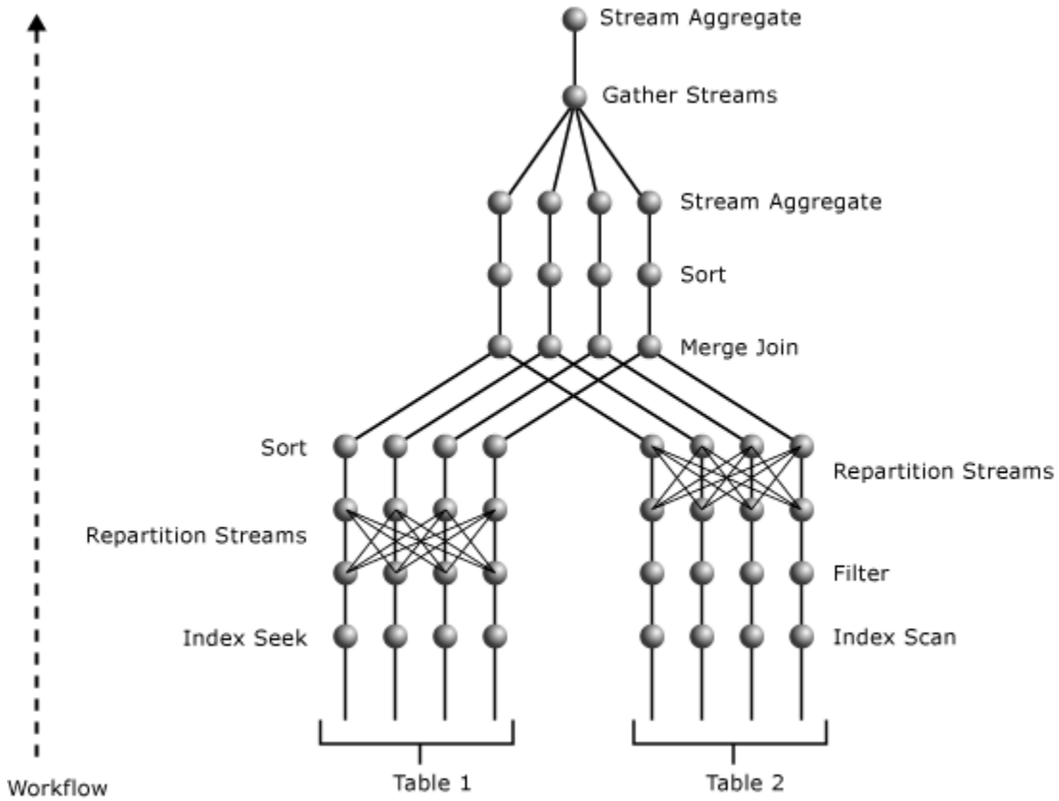
SQLCopy
```
CREATE INDEX l_order_dates_idx
  ON lineitem
    (l_orderkey, l_receiptdate, l_commitdate, l_shipdate)

CREATE UNIQUE INDEX o_datkeyopr_idx
  ON ORDERS
    (o_orderdate, o_orderkey, o_custkey, o_orderpriority)
```

Here is one possible parallel plan generated for the query previously shown:

Copy
```
|--Stream Aggregate(GROUP BY:([ORDERS].[o_orderpriority])
              DEFINE:([Expr1005]=COUNT(*)))
   |--Parallelism(Gather Streams, ORDER BY:
              ([ORDERS].[o_orderpriority] ASC))
       |--Stream Aggregate(GROUP BY:
              ([ORDERS].[o_orderpriority])
```

```
   DEFINE:([Expr1005]=Count(*)))
|--Sort(ORDER BY:([ORDERS].[o_orderpriority] ASC))
    |--Merge Join(Left Semi Join, MERGE:
   ([ORDERS].[o_orderkey])=
        ([LINEITEM].[l_orderkey]),
   RESIDUAL:([ORDERS].[o_orderkey]=
        [LINEITEM].[l_orderkey]))
        |--Sort(ORDER BY:([ORDERS].[o_orderkey] ASC))
        |    |--Parallelism(Repartition Streams,
          PARTITION COLUMNS:
          ([ORDERS].[o_orderkey]))
        |         |--Index Seek(OBJECT:
     ([tpcd1G].[dbo].[ORDERS].[O_DATKEYOPR_IDX]),
     SEEK:([ORDERS].[o_orderdate] >=
          Apr  1 2000 12:00AM AND
          [ORDERS].[o_orderdate] <
          Jul  1 2000 12:00AM) ORDERED)
       |--Parallelism(Repartition Streams,
    PARTITION COLUMNS:
    ([LINEITEM].[l_orderkey]),
    ORDER BY:([LINEITEM].[l_orderkey] ASC))
          |--Filter(WHERE:
          ([LINEITEM].[l_commitdate]<
          [LINEITEM].[l_receiptdate]))
               |--Index Scan(OBJECT:
([tpcd1G].[dbo].[LINEITEM].[L_ORDER_DATES_IDX]), ORDERED)
```

The illustration below shows a query plan executed with a degree of parallelism equal to 4 and involving a two-table join.



The parallel plan contains three parallelism operators. Both the Index Seek operator of the o_datkey_ptr index and the Index Scan operator of the l_order_dates_idx index are performed in parallel. This produces several exclusive streams. This can be determined from the nearest Parallelism operators above the Index Scan and Index Seek operators, respectively. Both are repartitioning the type of exchange. That is, they are just reshuffling data among the streams and producing the same number of streams on their output as they have on their input. This number of streams is equal to the degree of parallelism.

The parallelism operator above the l_order_dates_idx Index Scan operator is repartitioning its input streams using the value of L_ORDERKEY as a key. In this way, the same values of L_ORDERKEY end up in the same output stream. At the same time, output streams maintain the order on the L_ORDERKEY column to meet the input requirement of the Merge Join operator.

The parallelism operator above the Index Seek operator is repartitioning its input streams using the value of O_ORDERKEY. Because its input is not sorted on the O_ORDERKEY column values and this is the join column in the Merge Join operator, the Sort operator between the parallelism and Merge Join operators make sure that the input is sorted for the Merge Join operator on the join columns. The Sort operator, like the Merge Join operator, is performed in parallel.

The topmost parallelism operator gathers results from several streams into a single stream. Partial aggregations performed by the Stream Aggregate operator below the parallelism operator are then accumulated into a single SUM value for each different value of the O_ORDERPRIORITY in the Stream Aggregate operator above the parallelism operator. Because this plan has two exchange segments, with degree of parallelism equal to 4, it uses eight worker threads.

For more information on the operators used in this example, refer to the Showplan Logical and Physical Operators Reference.

Parallel Index Operations

The query plans built for the index operations that create or rebuild an index, or drop a clustered index, allow for parallel, multi-worker threaded operations on computers that have multiple microprocessors.

Note

Parallel index operations are only available in Enterprise Edition, starting with SQL Server 2008.

SQL Server uses the same algorithms to determine the degree of parallelism (the total number of separate worker threads to run) for index operations as it does for other queries. The maximum degree of parallelism for an index operation is subject to the max degree of parallelism server configuration option. You can override the max degree of parallelism value for individual index operations by setting the MAXDOP index option in the CREATE INDEX, ALTER INDEX, DROP INDEX, and ALTER TABLE statements.

When the SQL Server Database Engine builds an index execution plan, the number of parallel operations is set to the lowest value from among the following:

- The number of microprocessors, or CPUs in the computer.
- The number specified in the max degree of parallelism server configuration option.
- The number of CPUs not already over a threshold of work performed for SQL Server worker threads.

For example, on a computer that has eight CPUs, but where max degree of parallelism is set to 6, no more than six parallel worker threads are generated for an index operation. If five of the CPUs in the computer exceed the threshold of SQL Server work when an index execution plan is built, the execution plan specifies only three parallel worker threads.

The main phases of a parallel index operation include the following:

- A coordinating worker thread quickly and randomly scans the table to estimate the distribution of the index keys. The coordinating worker thread establishes the key boundaries that will create a number of key ranges equal to the degree of parallel operations, where each key range is estimated to cover similar numbers of rows. For example, if there are four million rows in the table and the degree of parallelism is 4, the coordinating worker thread will

determine the key values that delimit four sets of rows with 1 million rows in each set. If enough key ranges cannot be established to use all CPUs, the degree of parallelism is reduced accordingly.

- The coordinating worker thread dispatches a number of worker threads equal to the degree of parallel operations and waits for these worker threads to complete their work. Each worker thread scans the base table using a filter that retrieves only rows with key values within the range assigned to the worker thread. Each worker thread builds an index structure for the rows in its key range. In the case of a partitioned index, each worker thread builds a specified number of partitions. Partitions are not shared among worker threads.
- After all the parallel worker threads have completed, the coordinating worker thread connects the index subunits into a single index. This phase applies only to offline index operations.

Individual CREATE TABLE or ALTER TABLE statements can have multiple constraints that require that an index be created. These multiple index creation operations are performed in series, although each individual index creation operation may be a parallel operation on a computer that has multiple CPUs.

Distributed Query Architecture

Microsoft SQL Server supports two methods for referencing heterogeneous OLE DB data sources in Transact-SQL statements:

- Linked server names
  The system stored procedures sp_addlinkedserver and sp_addlinkedsrvlogin are used to give a server name to an OLE DB data source. Objects in these linked servers can be referenced in Transact-SQL statements using four-part names. For example, if a linked server name of DeptSQLSrvr is defined against another instance of SQL Server, the following statement references a table on that server:

  SQLCopy
  SELECT JobTitle, HireDate
  FROM DeptSQLSrvr.AdventureWorks2014.HumanResources.Employee;

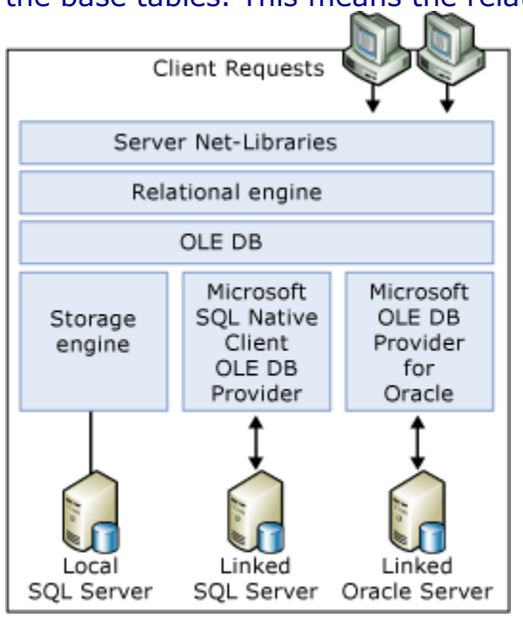
  The linked server name can also be specified in an OPENQUERY statement to open a rowset from the OLE DB data source. This rowset can then be referenced like a table in Transact-SQL statements.

- Ad hoc connector names
  For infrequent references to a data source, the OPENROWSET or OPENDATASOURCE functions are specified with the information needed to connect to the linked server. The rowset can then be referenced the same way a table is referenced in Transact-SQL statements:

  SQLCopy
  SELECT *
  FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',

'c:\MSOffice\Access\Samples\Northwind.mdb';'Admin';'';
Employees);

SQL Server uses OLE DB to communicate between the relational engine and the storage engine. The relational engine breaks down each Transact-SQL statement into a series of operations on simple OLE DB rowsets opened by the storage engine from the base tables. This means the relational engine can also open simple OLE DB rowsets on any OLE DB data source.



The relational engine uses the OLE DB application programming interface (API) to open the rowsets on linked servers, fetch the rows, and manage transactions.

For each OLE DB data source accessed as a linked server, an OLE DB provider must be present on the server running SQL Server. The set of Transact-SQL operations that can be used against a specific OLE DB data source depends on the capabilities of the OLE DB provider.

For each instance of SQL Server, members of the sysadmin fixed server role can enable or disable the use of ad-hoc connector names for an OLE DB provider using the SQL Server DisallowAdhocAccess property. When ad-hoc access is enabled, any user logged on to that instance can execute Transact-SQL statements containing ad-hoc connector names, referencing any data source on the network that can be accessed using that OLE DB provider. To control access to data sources, members of the sysadmin role can disable ad-hoc access for that OLE DB provider, thereby limiting users to only those data sources referenced by linked server names defined by the administrators. By default, ad-hoc access is enabled for the SQL Server OLE DB provider, and disabled for all other OLE DB providers.

Distributed queries can allow users to access another data source (for example, files, non-relational data sources such as Active Directory, and so on) using the security context of the Microsoft Windows account under which the SQL Server service is running. SQL Server impersonates the login appropriately for Windows logins; however, that is not possible for SQL Server logins. This can potentially allow a distributed query user to access another data source for which they do not have permissions, but the account under which the SQL Server service is running does have permissions.
Use sp_addlinkedsrvlogin to define the specific logins that are authorized to access the corresponding linked server. This control is not available for ad-hoc names, so use caution in enabling an OLE DB provider for ad-hoc access.

When possible, SQL Server pushes relational operations such as joins, restrictions, projections, sorts, and group by operations to the OLE DB data source. SQL Server does not default to scanning the base table into SQL Server and performing the relational operations itself. SQL Server queries the OLE DB provider to determine the level of SQL grammar it supports, and, based on that information, pushes as many relational operations as possible to the provider.

SQL Server specifies a mechanism for an OLE DB provider to return statistics indicating how key values are distributed within the OLE DB data source. This lets the SQL Server Query Optimizer better analyze the pattern of data in the data source against the requirements of each Transact-SQL statement, increasing the ability of the Query Optimizer to generate optimal execution plans.

Query Processing Enhancements on Partitioned Tables and Indexes

SQL Server 2008 improved query processing performance on partitioned tables for many parallel plans, changes the way parallel and serial plans are represented, and enhanced the partitioning information provided in both compile-time and run-time execution plans. This topic describes these improvements, provides guidance on how to interpret the query execution plans of partitioned tables and indexes, and provides best practices for improving query performance on partitioned objects.

Note

Until SQL Server 2014 (12.x), partitioned tables and indexes are supported only in the SQL Server Enterprise, Developer, and Evaluation editions.
Starting with SQL Server 2016 (13.x) SP1, partitioned tables and indexes are also supported in SQL Server Standard edition.

New Partition-Aware Seek Operation

In SQL Server, the internal representation of a partitioned table is changed so that the table appears to the query processor to be a multicolumn index with PartitionID as the leading column. PartitionID is a hidden computed column used internally to represent the ID of the partition containing a specific row. For example, assume the table T, defined as T(a, b, c), is partitioned on column a, and has a clustered index on column b. In SQL Server, this partitioned table is treated internally as a nonpartitioned table with the schema T(PartitionID, a, b, c) and a clustered index on the composite key (PartitionID, b). This allows the Query Optimizer to perform seek operations based on PartitionID on any partitioned table or index.

Partition elimination is now done in this seek operation.

In addition, the Query Optimizer is extended so that a seek or scan operation with one condition can be done on PartitionID (as the logical leading column) and possibly other index key columns, and then a second-level seek, with a different condition, can be done on one or more additional columns, for each distinct value that meets the qualification for the first-level seek operation. That is, this operation, called a skip scan, allows the Query Optimizer to perform a seek or scan operation based on one condition to determine the partitions to be accessed and a second-level index seek operation within that operator to return rows from these partitions that meet a different condition. For example, consider the following query.

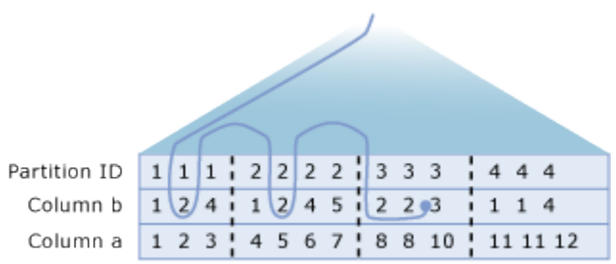
SQLCopy
SELECT * FROM T WHERE a < 10 and b = 2;

For this example, assume that table T, defined as T(a, b, c), is partitioned on column a, and has a clustered index on column b. The partition boundaries for table T are defined by the following partition function:

SQLCopy
CREATE PARTITION FUNCTION myRangePF1 (int) AS RANGE LEFT FOR VALUES (3, 7, 10);

To solve the query, the query processor performs a first-level seek operation to find every partition that contains rows that meet the condition T.a < 10. This identifies the partitions to be accessed. Within each partition identified, the processor then performs a second-level seek into the clustered index on column b to find the rows that meet the condition T.b = 2 and T.a < 10.

The following illustration is a logical representation of the skip scan operation. It shows table T with data in columns a and b. The partitions are numbered 1 through 4 with the partition boundaries shown by dashed vertical lines. A first-level seek operation to the partitions (not shown in the illustration) has determined that partitions 1, 2, and 3 meet the seek condition implied by the partitioning defined for the table and the predicate on column a. That is, T.a < 10. The path traversed by the second-level seek portion of the skip scan operation is illustrated by the curved line. Essentially, the skip scan operation seeks into each of these partitions for rows that meet the condition b = 2. The total cost of the skip scan operation is the same as that of three separate index seeks.



Displaying Partitioning Information in Query Execution Plans

The execution plans of queries on partitioned tables and indexes can be examined by using the Transact-SQL SET statements SET SHOWPLAN_XML or SET STATISTICS XML, or by using the graphical execution plan output in SQL Server Management Studio. For example, you can display the compile-time execution plan by clicking *Display Estimated Execution Plan* on the Query Editor toolbar and the run-time plan by clicking *Include Actual Execution Plan*.

Using these tools, you can ascertain the following information:

- The operations such as scans, seeks, inserts, updates, merges, and deletes that access partitioned tables or indexes.
- The partitions accessed by the query. For example, the total count of partitions accessed and the ranges of contiguous partitions that are accessed are available in run-time execution plans.
- When the skip scan operation is used in a seek or scan operation to retrieve data from one or more partitions.

Partition Information Enhancements

SQL Server provides enhanced partitioning information for both compile-time and run-time execution plans. Execution plans now provide the following information:

- An optional Partitioned attribute that indicates that an operator, such as a seek, scan, insert, update, merge, or delete, is performed on a partitioned table.
- A new SeekPredicateNew element with a SeekKeys subelement that includes PartitionID as the leading index key column and filter conditions that specify range seeks on PartitionID. The presence of two SeekKeys subelements indicates that a skip scan operation on PartitionID is used.
- Summary information that provides a total count of the partitions accessed. This information is available only in run-time plans.

To demonstrate how this information is displayed in both the graphical execution plan output and the XML Showplan output, consider the following query on the partitioned table fact_sales. This query updates data in two partitions.

SQLCopy
```
UPDATE fact_sales
SET quantity = quantity * 2
WHERE date_id BETWEEN 20080802 AND 20080902;
```

The following illustration shows the properties of the Clustered Index Seek operator in the runtime execution plan for this query. To view the definition of the fact_sales table and the partition definition, see "Example" in this topic.

Clustered Index Seek (Clustered)

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	967333
Actual Number of Rows	967333
Actual Number of Batches	0
Estimated Operator Cost	102.553 (5%)
Estimated I/O Cost	101.498
Estimated Subtree Cost	102.553
Estimated CPU Cost	1.05493
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows	958743
Estimated Number of Rows to be Read	958743
Estimated Row Size	23 B
Actual Rebinds	0
Actual Rewinds	0
Partitioned	True
Actual Partition Count	2
Ordered	True
Node ID	2

Object

[db_sales_test].[dbo].[fact_sales].[ci]

Output List

PtnId1000, Uniq1002, [db_sales_test].[dbo].[fact_sales].date_id, [db_sales_test].[dbo].[fact_sales].quantity

Seek Predicates

Seek Keys[1]: Start: PtnId1000 >= Scalar Operator (RangePartitionNew([@2],(1),(20080801),(20080901),(20081001),(20081101),(20081201),(20090101))), End: PtnId1000 <= Scalar Operator(RangePartitionNew([@3],(1),(20080801),(20080901),(20081001),(20081101),(20081201),(20090101))), Seek Keys[2]: Start: [db_sales_test].[dbo].[fact_sales].date_id >= Scalar Operator ([@2]), End: [db_sales_test].[dbo].[fact_sales].date_id <= Scalar Operator([@3])

Partitioned Attribute

When an operator such as an Index Seek is executed on a partitioned table or index, the Partitioned attribute appears in the compile-time and run-time plan and is set to True (1). The attribute does not display when it is set to False (0).

The Partitioned attribute can appear in the following physical and logical operators:
||| |--------|--------| |Table Scan|Index Scan| |Index Seek|Insert| |Update|Delete| |Merge||

As shown in the previous illustration, this attribute is displayed in the properties of the operator in which it is defined. In the XML Showplan output, this attribute appears as Partitioned="1" in the RelOp node of the operator in which it is defined.

New Seek Predicate

In XML Showplan output, the SeekPredicateNew element appears in the operator in which it is defined. It can contain up to two occurrences of the SeekKeys sub-element. The first SeekKeys item specifies the first-level seek operation at the partition ID level of the logical index. That is, this seek determines the partitions that must be accessed to satisfy the conditions of the query. The second SeekKeys item specifies the second-level seek portion of the skip scan operation that occurs within each partition identified in the first-level seek.

Partition Summary Information

In run-time execution plans, partition summary information provides a count of the partitions accessed and the identity of the actual partitions accessed. You can use this information to verify that the correct partitions are accessed in the query and that all other partitions are eliminated from consideration.

The following information is provided: Actual Partition Count, and Partitions Accessed.

Actual Partition Count is the total number of partitions accessed by the query.

Partitions Accessed, in XML Showplan output, is the partition summary information that appears in the new RuntimePartitionSummary element in RelOp node of the operator in which it is defined. The following example shows the contents of the RuntimePartitionSummary element, indicating that two total partitions are accessed (partitions 2 and 3).

Copy
```
<RunTimePartitionSummary>

    <PartitionsAccessed PartitionCount="2" >
```

```
        <PartitionRange Start="2" End="3" />

    </PartitionsAccessed>

</RunTimePartitionSummary>
```

Displaying Partition Information by Using Other Showplan Methods

The Showplan methods SHOWPLAN_ALL, SHOWPLAN_TEXT, and STATISTICS PROFILE do not report the partition information described in this topic, with the following exception. As part of the SEEK predicate, the partitions to be accessed are identified by a range predicate on the computed column representing the partition ID. The following example shows the SEEK predicate for a Clustered Index Seek operator. Partitions 2 and 3 are accessed, and the seek operator filters on the rows that meet the condition date_id BETWEEN 20080802 AND 20080902.

Copy

```
|--Clustered Index Seek(OBJECT:([db_sales_test].[dbo].[fact_sales].[ci]),

      SEEK:([PtnId1000] >= (2) AND [PtnId1000] \<= (3)

            AND [db_sales_test].[dbo].[fact_sales].[date_id] >= (20080802)

            AND [db_sales_test].[dbo].[fact_sales].[date_id] <= (20080902))

            ORDERED FORWARD)
```

Interpreting Execution Plans for Partitioned Heaps

A partitioned heap is treated as a logical index on the partition ID. Partition elimination on a partitioned heap is represented in an execution plan as a Table Scan operator with a SEEK predicate on partition ID. The following example shows the Showplan information provided:

Copy

```
|-- Table Scan (OBJECT: ([db].[dbo].[T]), SEEK: ([PtnId1001]=[Expr1011]) ORDERED FORWARD)
```

Interpreting Execution Plans for Collocated Joins

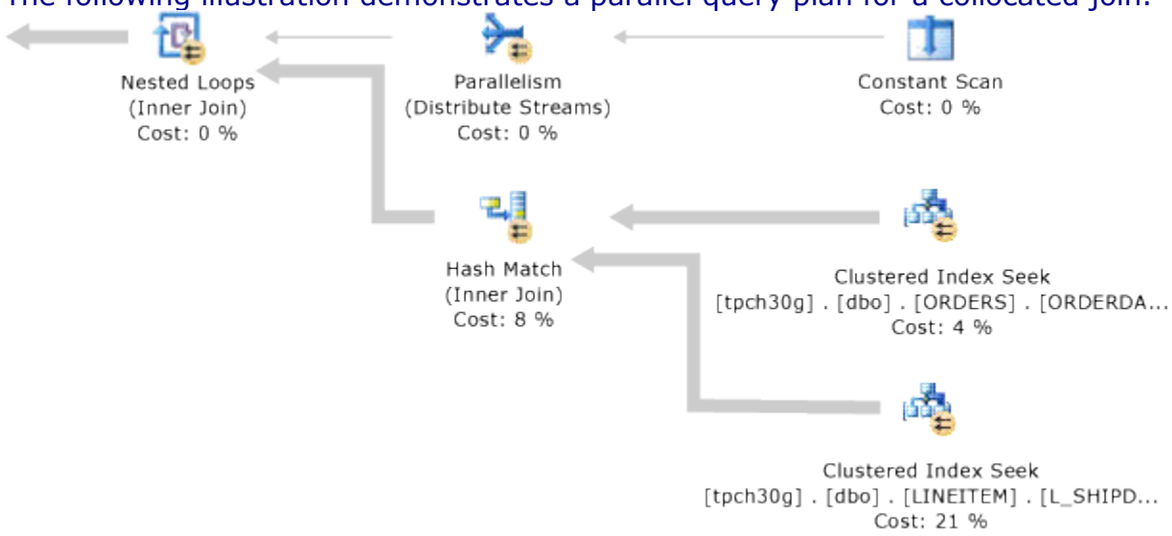
Join collocation can occur when two tables are partitioned using the same or equivalent partitioning function and the partitioning columns from both sides of the join are specified in the join condition of the query. The Query Optimizer can

generate a plan where the partitions of each table that have equal partition IDs are joined separately. Collocated joins can be faster than non-collocated joins because they can require less memory and processing time. The Query Optimizer chooses a non-collocated plan or a collocated plan based on cost estimates.

In a collocated plan, the Nested Loops join reads one or more joined table or index partitions from the inner side. The numbers within the Constant Scan operators represent the partition numbers.

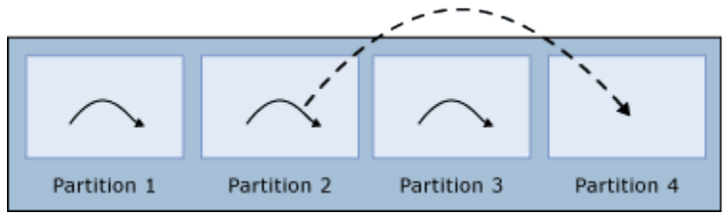
When parallel plans for collocated joins are generated for partitioned tables or indexes, a Parallelism operator appears between the Constant Scan and the Nested Loops join operators. In this case, multiple worker threads on the outer side of the join each read and work on a different partition.

The following illustration demonstrates a parallel query plan for a collocated join.

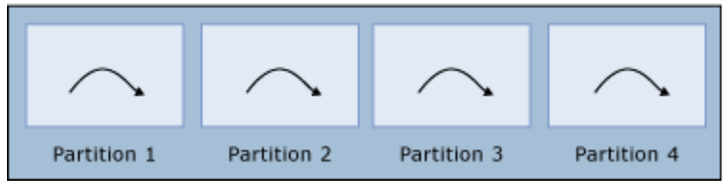


Parallel Query Execution Strategy for Partitioned Objects

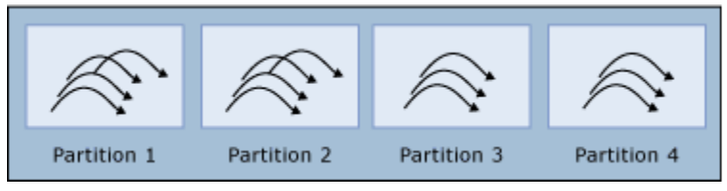
The query processor uses a parallel execution strategy for queries that select from partitioned objects. As part of the execution strategy, the query processor determines the table partitions required for the query and the proportion of worker threads to allocate to each partition. In most cases, the query processor allocates an equal or almost equal number of worker threads to each partition, and then executes the query in parallel across the partitions. The following paragraphs explain worker thread allocation in greater detail.



If the number of worker threads is less than the number of partitions, the query processor assigns each worker thread to a different partition, initially leaving one or more partitions without an assigned worker thread. When a worker thread finishes executing on a partition, the query processor assigns it to the next partition until each partition has been assigned a single worker thread. This is the only case in which the query processor reallocates worker threads to other partitions.
Shows worker thread reassigned after it finishes. If the number of worker threads is equal to the number of partitions, the query processor assigns one worker thread to each partition. When a worker thread finishes, it is not reallocated to another partition.



If the number of worker threads is greater than the number of partitions, the query processor allocates an equal number of worker threads to each partition. If the number of worker threads is not an exact multiple of the number of partitions, the query processor allocates one additional worker thread to some partitions in order to use all of the available worker threads. Note that if there is only one partition, all worker threads will be assigned to that partition. In the diagram below, there are four partitions and 14 worker threads. Each partition has 3 worker threads assigned, and two partitions have an additional worker thread, for a total of 14 worker thread assignments. When a worker thread finishes, it is not reassigned to another partition.



Although the above examples suggest a straightforward way to allocate worker threads, the actual strategy is more complex and accounts for other variables that occur during query execution. For example, if the table is partitioned and has a clustered index on column A and a query has the predicate clause WHERE A IN (13, 17, 25), the query processor will allocate one or more worker threads to each of these three seek values (A=13, A=17, and A=25) instead of each table partition. It is only

necessary to execute the query in the partitions that contain these values, and if all of these seek predicates happen to be in the same table partition, all of the worker threads will be assigned to the same table partition.

To take another example, suppose that the table has four partitions on column A with boundary points (10, 20, 30), an index on column B, and the query has a predicate clause WHERE B IN (50, 100, 150). Because the table partitions are based on the values of A, the values of B can occur in any of the table partitions. Thus, the query processor will seek for each of the three values of B (50, 100, 150) in each of the four table partitions. The query processor will assign worker threads proportionately so that it can execute each of these 12 query scans in parallel.

TABLE 2

Table partitions based on column A	Seeks for column B in each table partition
Table Partition 1: A < 10	B=50, B=100, B=150
Table Partition 2: A >= 10 AND A < 20	B=50, B=100, B=150
Table Partition 3: A >= 20 AND A < 30	B=50, B=100, B=150
Table Partition 4: A >= 30	B=50, B=100, B=150

Best Practices

To improve the performance of queries that access a large amount of data from large partitioned tables and indexes, we recommend the following best practices:

- Stripe each partition across many disks. This is especially relevant when using spinning disks.
- When possible, use a server with enough main memory to fit frequently accessed partitions or all partitions in memory to reduce I/O cost.
- If the data you query will not fit in memory, compress the tables and indexes. This will reduce I/O cost.
- Use a server with fast processors and as many processor cores as you can afford, to take advantage of parallel query processing capability.
- Ensure the server has sufficient I/O controller bandwidth.
- Create a clustered index on every large partitioned table to take advantage of B-tree scanning optimizations.
- Follow the best practice recommendations in the white paper, The Data Loading Performance Guide, when bulk loading data into partitioned tables.

Example

The following example creates a test database containing a single table with seven partitions. Use the tools described previously when executing the queries in this example to view partitioning information for both compile-time and run-time plans.

Note

This example inserts more than 1 million rows into the table. Running this example may take several minutes depending on your hardware. Before executing this example, verify that you have more than 1.5 GB of disk space available.

SQLCopy

```
USE master;
GO
IF DB_ID (N'db_sales_test') IS NOT NULL
    DROP DATABASE db_sales_test;
GO
CREATE DATABASE db_sales_test;
GO
USE db_sales_test;
GO
CREATE PARTITION FUNCTION [pf_range_fact](int) AS RANGE RIGHT FOR VALUES
(20080801, 20080901, 20081001, 20081101, 20081201, 20090101);
GO
CREATE PARTITION SCHEME [ps_fact_sales] AS PARTITION [pf_range_fact]
ALL TO ([PRIMARY]);
GO
CREATE TABLE fact_sales(date_id int, product_id int, store_id int,
    quantity int, unit_price numeric(7,2), other_data char(1000))
ON ps_fact_sales(date_id);
GO
CREATE CLUSTERED INDEX ci ON fact_sales(date_id);
GO
PRINT 'Loading...';
SET NOCOUNT ON;
DECLARE @i int;
SET @i = 1;
WHILE (@i<1000000)
BEGIN
    INSERT INTO fact_sales VALUES(20080800 + (@i%30) + 1, @i%10000, @i%200, RAND() * 25, (@i%3) + 1, '');
    SET @i += 1;
END;
GO
```

```
DECLARE @i int;
SET @i = 1;
WHILE (@i<10000)
BEGIN
   INSERT INTO fact_sales VALUES(20080900 + (@i%30) + 1, @i%10000, @i%200, RAND() * 25, (@i%3) + 1, '');
   SET @i += 1;
END;
PRINT 'Done.';
GO
-- Two-partition query.
SET STATISTICS XML ON;
GO
SELECT date_id, SUM(quantity*unit_price) AS total_price
FROM fact_sales
WHERE date_id BETWEEN 20080802 AND 20080902
GROUP BY date_id ;
GO
SET STATISTICS XML OFF;
GO
-- Single-partition query.
SET STATISTICS XML ON;
GO
SELECT date_id, SUM(quantity*unit_price) AS total_price
FROM fact_sales
WHERE date_id BETWEEN 20080801 AND 20080831
GROUP BY date_id;
GO
SET STATISTICS XML OFF;
GO
```